Manshaa Kakani | BE: II Year | Information Technology (IT): A

Operating System Lab Assignment: IV

Table of Contents

[ Aim of assignment 2](#_Toc194930091)

[ To Perform 2](#_Toc194930092)

[ To Submit 2](#_Toc194930093)

[I. Introduction 2](#_Toc194930094)

[Definition of Linux System Calls 2](#_Toc194930095)

[Objective of the Study 2](#_Toc194930096)

[Importance of System Calls 3](#_Toc194930097)

[II. Categorization of System Calls 3](#_Toc194930098)

[1. Process Management System Calls 3](#_Toc194930099)

[2. File Management System Calls 9](#_Toc194930100)

[3. Device Management System Calls 15](#_Toc194930101)

[4. Network Management System Calls 22](#_Toc194930102)

[5.System Information Management System Calls 30](#_Toc194930103)

[III. Summary of Findings 35](#_Toc194930104)

[IV. Conclusion 36](#_Toc194930105)

# Aim of assignment

To study and learn about various system calls

# To Perform

Comprehensive study of different categories of Linux system calls, categorized as

1. Process Management System calls: fork(), exec(), wait(), exit().

2. File Management System calls: open(), read(), write(), close().

3. Device Management System calls: read(), write(), ioctl(), select().

4. Network Management System calls: socket(), connect(), send(), recv().

5. System Information Management System calls: getpid(), getuid(), gethostname(), sysinfo().

# To Submit

Write up for the exhaustive study of the above-mentioned system call categories with their examples.

# I. Introduction

## Definition of Linux System Calls

In any modern operating system, system calls serve as the critical interface that connects user-level applications to the kernel. They are the mechanism through which programs request essential services from the OS, such as accessing hardware, managing processes, or performing file operations. This structured interaction ensures that programs can utilize system resources without directly interacting with the kernel, maintaining security, stability, and efficiency.

System calls can be seen as the backbone of operating systems, providing a safe and standardized gateway for communication between user programs and the underlying hardware. The kernel, as the core of the OS, controls resources and enforces rules, and system calls ensure that this control is upheld while allowing programs to perform their tasks seamlessly.

Their significance is particularly seen in system programming and backend development, where understanding system calls is key to designing robust and efficient software. By providing this essential interface, system calls enable developers to build powerful applications.

## Objective of the Study

Building on the importance of Linux system calls as the backbone of communication between applications and the kernel, this study aims to dive deeper by categorizing and analysing them systematically.

By organizing these system calls into distinct functional groups, we can break down their complexity and understand their individual roles in managing key operating system tasks.

The scope of this study encompasses the following five critical categories:

1. **Process Management** – Handling the creation, synchronization, and termination of processes.
2. **File Management** – Facilitating file operations like reading, writing, and managing file descriptors.
3. **Device Management** – Managing communication and operations with hardware devices.
4. **Network Management** – Enabling data exchange over networks using sockets.
5. **System Information Management** – Retrieving essential details about the system and its state.

## Importance of System Calls

Linux system calls enable seamless interaction between user applications and the operating system, providing controlled access to critical resources such as CPU, memory, files, devices, and networks.

For developers, understanding and utilizing system calls effectively leads to better system performance, improved resource management, and robust software design.

In backend development, system calls play a vital role in optimizing processes like database queries, file handling, and network communication. For instance, when building a backend server, system calls such as socket() and send() allow applications to establish and manage connections with clients over a network.

A practical real-life example can be seen in a web server. When a client sends an HTTP request, the server uses system calls like recv() to read the request and send() to transmit the response. At the same time, fork() might be used to create child processes to handle multiple client connections concurrently.

# II. Categorization of System Calls

Building on the importance and objectives outlined earlier, it is essential to examine Linux system calls through structured categories to better understand their specific functionalities. The five categories are grouped based on the nature of the tasks they perform and the domains they operate within:

### 1. Process Management System Calls

Process management system calls directly facilitate the creation, execution, synchronization, and termination of processes. These system calls act as an interface for managing the kernel’s process table, which stores essential metadata about each process, including its state, priority, and resource allocations.

Processes in Linux operate within a hierarchical structure, where each process has a parent and potentially multiple children. Process management system calls enable developers to manipulate this hierarchy, allowing for operations such as forking a new child process, replacing the process image with a new executable, or terminating a process and reclaiming its resources.

By using these system calls, tasks such as multitasking, load balancing, and inter-process communication (IPC) become possible. The kernel ensures that all processes share system resources fairly while maintaining isolation and security.

#### fork()

##### Functionality

The fork() system call is used to create a new process, called the child process, by duplicating the parent process. The child process inherits most of the attributes of the parent process, including the process environment, open file descriptors, and code segment. However, the child process receives a unique process ID (PID). After the call, two processes run simultaneously: the parent and the child.

##### Parameters

The fork() system call does not require any parameters and returns an integer value:

* **0** in the child process.
* **PID of the child** in the parent process.
* **-1** if the creation of the child process fails.

##### Behaviour

* The parent process continues execution after the fork() call, and the child process starts execution from the point of the fork() call.
* The return value is critical to differentiate between the two processes.

##### Example

#include <stdio.h>

#include <unistd.h>

int main() {

pid\_t pid = fork(); // Create a new process

if (pid == 0) {

// This block is executed by the child process

printf("Child process: PID = %d\n", getpid());

} else if (pid > 0) {

// This block is executed by the parent process

printf("Parent process: PID = %d\n", getpid());

} else {

// Fork failed

printf("Error: Failed to create child process\n");

}

return 0;

}

##### Output (example):

Parent process: PID = 1234

Child process: PID = 1235

##### Practical Use Cases

1. Multitasking: fork() is commonly used to create child processes for concurrent execution. For instance, a server handling multiple client requests can fork a new process for each client, ensuring independent and parallel processing.
2. Process Isolation: By creating a child process, developers can isolate tasks. For example, executing risky operations in a child process can protect the parent process from instability.
3. Background Tasks: Child processes created by fork() are often used to run background tasks like logging, monitoring, or executing periodic operations.

#### exec()

##### Functionality

The exec() family of system calls is used to replace the current process image with a new one. This means that the calling process is transformed into an entirely new program, effectively overwriting its code, stack, and data segments. However, the process ID (PID) remains the same, as it is still the same process, albeit with a new execution context.

exec() is part of a family of functions, such as execl(), execp(), execv(), etc., which differ in how they accept arguments and environment variables. It is primarily used in conjunction with fork()—after a new process is forked, exec() can be used to load a new executable into the child process.

##### Parameters

The parameters of exec() vary depending on the specific variant used. For example, execvp() takes:

* The path to the executable (e.g., /bin/ls).
* An array of arguments, where the first argument is typically the program name.

##### Behaviour

* When exec() is called, the process image is replaced, and execution begins from the entry point of the new program.
* The function does not return unless an error occurs (e.g., the specified executable cannot be found or executed).

##### Example

#include <stdio.h>

#include <unistd.h>

int main() {

char \*args[] = {"ls", "-l", NULL}; // Arguments to pass to the new program

printf("Before exec()\n");

execvp("ls", args); // Replace current process with 'ls -l'

// This will only execute if execvp fails

perror("execvp failed");

return 1;

}

##### Output (example):

Before exec()

<Output of 'ls -l'>

##### Practical Use Cases

1. Program Execution in a New Context: Often used in shell programs or script interpreters to execute user commands by replacing the current process with the specified program.
2. Custom Initialization: After a process is forked, exec() is commonly used to initialize a new environment by loading a specific executable, such as a server process or a worker thread.
3. Command-Line Utilities: Backend systems often use exec() to integrate various command-line utilities as part of automated workflows or pipelines.

#### wait()

##### Functionality

The wait() system call allows a parent process to pause its execution until one of its child processes terminates. This is crucial for process synchronization, as it ensures that the parent process can manage child processes effectively. Once the child process finishes, wait() returns the child’s termination status to the parent, enabling the parent to handle post-termination tasks or cleanup.

##### Parameters

* status: A pointer to an integer where the termination status of the child process will be stored. If the status is not needed, NULL can be passed.

##### Behaviour

* If a parent process calls wait() and has no child processes, the call fails immediately.
* If child processes are running, the parent process suspends execution until one child process terminates or a signal interrupts it.

##### Example

#include <stdio.h>

#include <sys/types.h>

#include <sys/wait.h>

#include <unistd.h>

int main() {

pid\_t pid = fork();

if (pid == 0) {

// Child process

printf("Child process: PID = %d\n", getpid());

sleep(2); // Simulate some work

printf("Child process exiting...\n");

return 42; // Exit status

} else if (pid > 0) {

// Parent process

int status;

wait(&status); // Wait for child to finish

printf("Parent process: Child exited with status %d\n", WEXITSTATUS(status));

} else {

// Fork failed

perror("Fork failed");

}

return 0;

}

##### Output (example):

Child process: PID = 1234

Child process exiting...

Parent process: Child exited with status 42

##### Practical Use Cases

1. Process Synchronization: The wait() system call ensures that parent processes do not proceed until their child processes have completed, making it essential for maintaining order in workflows.
2. Resource Management: By waiting for child processes to terminate, wait() allows the parent process to reclaim resources such as memory and file descriptors allocated to the child.
3. Error Handling: The termination status returned by wait() enables the parent process to detect errors in the child’s execution and take corrective actions if needed.

#### exit()

##### Functionality

The exit() system call is used to terminate a process and return a status code to the operating system. When a process exits, the kernel cleans up all resources allocated to it, such as memory, file descriptors, and other process-specific data. The exit status returned can be used by the parent process (via wait()) to understand whether the child process terminated successfully or encountered an error.

##### Parameters

* status: An integer value that indicates the termination status of the process. This value is typically used to signal success (e.g., 0) or failure (non-zero values).

##### Behaviour

* Once exit() is called, the process is terminated immediately.
* The kernel removes the process from the process table and releases all allocated resources.
* The exit status is returned to the parent process, if applicable.

##### Example

#include <stdio.h>

#include <stdlib.h>

int main() {

printf("Process is about to exit...\n");

exit(0); // Terminate process with status code 0 (success)

// This line will not execute since exit() ends the process

printf("This will never be printed.\n");

}

##### Output:

Process is about to exit...

##### Practical Use Cases

1. Graceful Termination: exit() is often used to end processes after completing tasks, ensuring resources are freed up properly and returning meaningful status codes to parent processes or error-handling mechanisms.
2. Error Reporting: In programs where errors need to be flagged, a non-zero exit code can signal specific issues, which can then be interpreted by scripts or monitoring tools.
3. System Shutdowns and Cleanup: exit() can be used at the end of programs to ensure all allocated resources, such as open files or network connections, are released before termination.

### 2. File Management System Calls

File management system calls are pivotal for performing Input/Output (I/O) operations on files. These calls enable processes to interact with the file system, facilitating operations such as opening, reading, writing, and closing files. They abstract the complexities of low-level file handling, providing developers with a standardized interface to work with files without needing to manage hardware-specific details.

In Linux, files are treated as streams of bytes, regardless of whether they represent text, binary data, or even hardware devices (like /dev/null). File system calls allow developers to interact with these streams efficiently, ensuring proper resource allocation and error management during file operations.

By utilizing these calls, backend systems can handle critical tasks, such as logging application events, reading configuration files, or writing output to external storage devices. Understanding and correctly implementing file I/O system calls is essential for designing reliable and scalable backend solutions.

#### open()

##### Functionality

The open() system call is used to open a file and obtain a file descriptor for it. The file descriptor is a unique identifier through which the file can be accessed for subsequent operations like read(), write(), or close().

##### Parameters

1. pathname: Specifies the file's location (e.g., "file.txt").
2. flags: Define the mode of access, such as:
   * O\_RDONLY (read-only),
   * O\_WRONLY (write-only), or
   * O\_RDWR (read/write). Additional flags, such as O\_CREAT, can be used to create the file if it doesn’t exist.
3. mode (optional): Sets file permissions if the file is created (e.g., 0666 for full read and write access).

##### Behaviour

* If successful, open() returns a non-negative file descriptor.
* If it fails (e.g., the file is missing and O\_CREAT isn’t specified), it returns -1.

##### Example

#include <fcntl.h>

#include <unistd.h>

#include <stdio.h>

int main() {

int fd = open("example.txt", O\_RDWR | O\_CREAT, 0666); // Open or create file

if (fd < 0) {

perror("Failed to open file");

return 1;

}

printf("File opened successfully: FD = %d\n", fd);

close(fd); // Close the file descriptor

return 0;

}

##### Output (example):

File opened successfully: FD = 3

##### Practical Use Cases

1. File Creation: Automatically create a log file if it doesn’t exist, enabling dynamic logging.
2. Access Control: Specify read-only access to configuration files to prevent unintended modifications.

#### read()

##### Functionality

The read() system call is used to read data from an open file into a buffer. It begins reading from the current file offset and advances the offset by the number of bytes successfully read. This operation is fundamental for file handling and enables processes to extract data for further processing.

##### Parameters

1. fd: The file descriptor obtained from a previous open() call.
2. buf: A pointer to a buffer where the data read from the file will be stored.
3. count: The maximum number of bytes to read.

##### Behavior

* Return Value: The actual number of bytes successfully read.
* Returns 0 when the end of the file (EOF) is reached.
* Returns -1 if an error occurs, such as when the file descriptor is invalid.

##### Example

#include <fcntl.h>

#include <unistd.h>

#include <stdio.h>

int main() {

char buffer[128]; // Buffer to store file data

int fd = open("example.txt", O\_RDONLY); // Open file for reading

if (fd < 0) {

perror("Failed to open file");

return 1;

}

ssize\_t bytesRead = read(fd, buffer, sizeof(buffer) - 1); // Read data

if (bytesRead < 0) {

perror("Failed to read file");

} else {

buffer[bytesRead] = '\0'; // Null-terminate the data read

printf("Data read from file: %s\n", buffer);

}

close(fd); // Close the file descriptor

return 0;

}

##### Output (example):

Data read from file: Content of the file...

##### Practical Use Cases

1. Configuration File Parsing: Reading application settings from configuration files during initialization.
2. Data Processing: Extracting and processing information stored in external files, such as logs or structured data files.
3. Dynamic Operations: Using the read operation to retrieve input data for dynamic processing, such as user-provided inputs stored in temporary files.

#### write()

##### Functionality

The write() system call is used to write data from a buffer to an open file. It begins writing at the current file offset and updates the offset by the number of bytes successfully written. This operation is essential for storing data in files and ensuring proper handling of output streams.

##### Parameters

1. fd: The file descriptor obtained from the open() system call.
2. buf: A pointer to the buffer containing the data to be written.
3. count: The number of bytes to write.

##### Behavior

* Return Value: The actual number of bytes successfully written, which may be less than count due to factors like insufficient disk space.
* Returns -1 if an error occurs, such as an invalid file descriptor or write permissions.

##### Example

#include <fcntl.h>

#include <unistd.h>

#include <stdio.h>

int main() {

int fd = open("output.txt", O\_WRONLY | O\_CREAT, 0666); // Open or create file

if (fd < 0) {

perror("Failed to open file");

return 1;

}

const char \*data = "File Management System Call Example";

ssize\_t bytesWritten = write(fd, data, sizeof(data) - 1); // Write data to file

if (bytesWritten < 0) {

perror("Failed to write to file");

} else {

printf("Data written successfully: %ld bytes\n", bytesWritten);

}

close(fd); // Close the file descriptor

return 0;

}

##### Output (example):

Data written successfully: 34 bytes

##### Practical Use Cases

1. Log Writing: Writing logs to a file for debugging or monitoring application events in backend systems.
2. Report Generation: Storing generated output, such as processed data or results, in files for future use.
3. Communication via Files: Transferring data between processes using temporary files in inter-process communication (IPC).

#### **close()**

##### Functionality

The close() system call is used to terminate the association between a file descriptor and the file it represents. By calling close(), the kernel releases the resources allocated to the file descriptor, such as memory and file system locks, ensuring that no resource leakage occurs. Once closed, the file descriptor becomes invalid and can no longer be used for further operations.

##### Parameters

* fd: The file descriptor to be closed, which must have been obtained from a prior open() or similar call.

##### Behavior

* Return Value: Returns 0 on successful closure of the file descriptor.
* Returns -1 if an error occurs, such as attempting to close an already-closed or invalid file descriptor.

##### Example

#include <fcntl.h>

#include <unistd.h>

#include <stdio.h>

int main() {

int fd = open("example.txt", O\_RDWR | O\_CREAT, 0666); // Open or create file

if (fd < 0) {

perror("Failed to open file");

return 1;

}

printf("File descriptor opened successfully: FD = %d\n", fd);

if (close(fd) == 0) {

printf("File descriptor closed successfully.\n");

} else {

perror("Failed to close file descriptor");

}

return 0;

}

##### Output (example):

File descriptor opened successfully: FD = 3

File descriptor closed successfully.

##### Practical Use Cases

1. Resource Management: Ensuring that file descriptors are closed properly after their intended use to prevent resource leaks in applications.
2. File System Integrity: Closing files securely avoids corruption or inconsistency, especially in cases where multiple processes access the same file.
3. Application Stability: Regularly using close() minimizes the risk of exhausting file descriptors in long-running applications.

### 3. Device Management System Calls

Device management system calls facilitate interaction between processes and hardware devices, such as printers, keyboards, or monitors. They allow applications to perform input/output (I/O) operations on devices, configure device settings, and monitor device activity. In Linux, devices are often represented as files under the /dev directory, and these system calls provide an abstraction layer for device communication.

Device management is essential for handling hardware efficiently in backend systems and system-level programming. By using these calls, processes can access and control devices securely and reliably, avoiding direct interaction with hardware that might compromise stability.

#### read()

##### Functionality

The read() system call is essential for retrieving input data from a device into a buffer. When used for devices, read() is often implemented to capture data streams in a manner specific to the device's functionality. For example, reading from a keyboard device would return user-typed input, while reading from a sensor device might provide raw sensor data. The kernel ensures that data from the device is safely transferred to the user-provided buffer.

##### Parameters

1. fd: The file descriptor of the device, typically obtained by opening the device file in /dev (e.g., /dev/tty for a terminal).
2. buf: A pointer to a buffer where data from the device will be stored.
3. count: The maximum number of bytes to read from the device.

##### Behavior

* Returns the number of bytes actually read, which could be less than count if fewer bytes are available.
* Returns 0 if the end of the device's readable data stream is reached (applicable for special devices or files).
* Returns -1 if an error occurs, such as insufficient device permissions.

##### Example

#include <fcntl.h>

#include <unistd.h>

#include <stdio.h>

int main() {

char buffer[128]; // Buffer to store device data

int fd = open("/dev/tty", O\_RDONLY); // Open terminal device for reading

if (fd < 0) {

perror("Failed to open device");

return 1;

}

printf("Type something: ");

ssize\_t bytesRead = read(fd, buffer, sizeof(buffer) - 1); // Read input

if (bytesRead < 0) {

perror("Failed to read from device");

} else {

buffer[bytesRead] = '\0'; // Null-terminate the input

printf("\nData read: %s\n", buffer);

}

close(fd); // Close the device file descriptor

return 0;

}

##### Output (example):

Type something: Hello, device management!

Data read: Hello, device management!

##### Practical Use Cases

1. Keyboard Input: Capturing user input from a keyboard device for interactive applications or command-line tools.
2. Sensor Data Collection: Reading raw data streams from devices like temperature or motion sensors in IoT systems.
3. Device Monitoring: Fetching status or real-time data from hardware devices like serial ports or communication interfaces.

#### write()

##### Functionality

The write() system call is used to send data from a buffer to a device. It writes data starting at the current file offset, advancing the offset by the number of bytes successfully written. For devices, write() enables applications to send output or commands, depending on the specific functionality of the device.

##### Parameters

1. fd: The file descriptor of the device, obtained through open().
2. buf: A pointer to the buffer containing the data to be written to the device.
3. count: The number of bytes to write from the buffer to the device.

##### Behavior

* Returns the number of bytes actually written, which may be less than count if the device has limitations or constraints.
* Returns -1 if an error occurs, such as insufficient permissions or device unavailability.

##### Example

#include <fcntl.h>

#include <unistd.h>

#include <stdio.h>

int main() {

int fd = open("/dev/tty", O\_WRONLY); // Open terminal device for writing

if (fd < 0) {

perror("Failed to open device");

return 1;

}

const char \*message = "Hello, Device Management!";

ssize\_t bytesWritten = write(fd, message, sizeof(message) - 1); // Write to device

if (bytesWritten < 0) {

perror("Failed to write to device");

} else {

printf("Data written successfully: %ld bytes\n", bytesWritten);

}

close(fd); // Close the device file descriptor

return 0;

}

##### Output (example):

Data written successfully: 24 bytes

##### Practical Use Cases

1. Command Execution: Sending commands to devices like printers or terminals to perform specific operations.
2. Data Transmission: Writing data to communication devices such as serial ports for real-time transmission.
3. Device Interaction: Sending configuration settings or operational instructions to hardware devices like sensors or actuators.

#### ioctl()

##### Functionality

The ioctl() (Input/Output Control) system call provides a mechanism for performing device-specific operations or for configuring devices that go beyond basic read() and write() operations. It serves as a versatile interface for controlling hardware settings, querying device attributes, or manipulating device parameters. The exact behavior of ioctl() is device-dependent, as different devices implement their own specific commands.

##### Parameters

1. fd: The file descriptor for the device, obtained through open().
2. request: A device-specific request code that specifies the operation to perform. These codes are typically defined in header files like <sys/ioctl.h>.
3. argp: A pointer to data or a structure that the device driver uses for the requested operation. This can be an input parameter, an output parameter, or both.

##### Behavior

* Returns 0 on success for most operations.
* Returns -1 if an error occurs, such as an invalid request code or a lack of necessary permissions.

##### Example

#include <sys/ioctl.h>

#include <stdio.h>

#include <unistd.h>

int main() {

struct winsize ws;

if (ioctl(STDOUT\_FILENO, TIOCGWINSZ, &ws) == -1) { // Get window size

perror("Failed to get window size");

return 1;

}

printf("Rows: %d, Columns: %d\n", ws.ws\_row, ws.ws\_col); // Display size

return 0;

}

##### Output (example):

Rows: 24, Columns: 80

##### Practical Use Cases

1. Device Configuration: Adjusting device parameters, such as baud rates for serial ports or brightness levels for display devices.
2. Querying Device Attributes: Retrieving specific information, like the size of a storage device or the state of a network interface.
3. Custom Device Operations: Implementing unique device control functionalities that are not covered by standard system calls, such as enabling special modes or diagnostics.

#### select()

##### Functionality

The select() system call is used to monitor multiple file descriptors, such as those associated with devices, files, or sockets, and determine which ones are ready for I/O operations (e.g., read(), write(), or exceptional conditions). This is particularly useful in scenarios where a program needs to manage multiple input or output streams without blocking on a single descriptor.

##### Parameters

1. nfds: The range of file descriptors to monitor (i.e., the highest file descriptor number plus one).
2. readfds: A set of file descriptors to monitor for readability (can be NULL if not used).
3. writefds: A set of file descriptors to monitor for writability (can be NULL if not used).
4. exceptfds: A set of file descriptors to monitor for exceptional conditions (can be NULL if not used).
5. timeout: A pointer to a struct timeval specifying the maximum wait time. If NULL, select() waits indefinitely.

##### Behavior

* Returns the total number of file descriptors ready for the specified operations.
* Returns 0 if the timeout expires with no descriptors becoming ready.
* Returns -1 if an error occurs.

##### Example

#include <sys/select.h>

#include <unistd.h>

#include <stdio.h>

int main() {

fd\_set readfds;

struct timeval timeout;

FD\_ZERO(&readfds); // Initialize the file descriptor set

FD\_SET(STDIN\_FILENO, &readfds); // Add standard input (terminal) to the set

timeout.tv\_sec = 10; // Set timeout to 10 seconds

timeout.tv\_usec = 0;

printf("Waiting for input (10 seconds)...\n");

int ret = select(STDIN\_FILENO + 1, &readfds, NULL, NULL, &timeout);

if (ret == -1) {

perror("select() failed");

} else if (ret == 0) {

printf("Timeout occurred! No input.\n");

} else {

if (FD\_ISSET(STDIN\_FILENO, &readfds)) {

printf("Input is ready! Reading...\n");

char buffer[128];

ssize\_t bytesRead = read(STDIN\_FILENO, buffer, sizeof(buffer) - 1);

if (bytesRead > 0) {

buffer[bytesRead] = '\0';

printf("You typed: %s\n", buffer);

}

}

}

return 0;

}

##### Output (example):

Waiting for input (10 seconds)...

(Input during timeout period): You typed: Hello, world!

Or, if no input is provided:

Timeout occurred! No input.

##### Practical Use Cases

1. Multiplexing I/O Streams: Managing multiple input/output streams, such as sockets in a server application handling multiple client connection.
2. Real-Time Monitoring: Watching multiple device descriptors for readiness, ensuring non-blocking operations in interactive applications.
3. Timeout Handling: Introducing time-bound waits to avoid indefinite blocking when monitoring descriptors.

### 4. Network Management System Calls

Network management system calls enable processes to establish, manage, and communicate over network connections. These calls provide the foundation for all networked operations, such as sending and receiving data across the internet or a local network. They are integral to backend development, especially in building applications like servers, clients, and distributed systems.

These system calls abstract the complexities of low-level networking, offering a simplified interface for creating sockets, establishing connections, and performing data transfer operations. This abstraction ensures developers can focus on higher-level application logic without delving into hardware-specific network communication.

#### socket()

##### Functionality

The socket() system call is used to create a new socket, which serves as an endpoint for network communication. A socket allows processes to send and receive data over a network, enabling communication between applications on the same device or across the internet. Sockets can be configured for different protocols (e.g., TCP, UDP) and domains (e.g., IPv4, IPv6).

##### Parameters

1. domain: Specifies the communication domain (protocol family), such as:
   * AF\_INET: IPv4 Internet protocols.
   * AF\_INET6: IPv6 Internet protocols.
   * AF\_UNIX: Local communication within the same host.
2. type: Specifies the socket type, such as:
   * SOCK\_STREAM: A reliable, connection-oriented stream (TCP).
   * SOCK\_DGRAM: A datagram-oriented, connectionless socket (UDP).
3. protocol: Specifies the protocol to be used with the socket. Typically set to 0, which automatically selects the default protocol for the chosen domain and type.

##### Behavior

* Returns a file descriptor representing the created socket if successful.
* Returns -1 if an error occurs (e.g., invalid arguments or insufficient permissions).

##### Example

#include <sys/socket.h>

#include <netinet/in.h>

#include <stdio.h>

int main() {

int sockfd = socket(AF\_INET, SOCK\_STREAM, 0); // Create IPv4 TCP socket

if (sockfd < 0) {

perror("Failed to create socket");

return 1;

}

printf("Socket created successfully: FD = %d\n", sockfd);

// Close the socket when done

close(sockfd);

return 0;

}

##### Output (example):

Socket created successfully: FD = 3

##### Practical Use Cases

1. Client-Server Communication: Creating sockets is the first step in establishing connections between clients and servers in backend applications.
2. Network Protocol Implementations: Sockets are used to implement network protocols such as HTTP, FTP, or custom communication protocols.
3. Real-Time Applications: They enable real-time data exchange, such as in messaging apps, online gaming, or video streaming platforms.

#### connect()

##### Functionality

The connect() system call establishes a connection between a client and a server socket. It is used primarily on client-side sockets to initiate communication with a server. The connect() call binds the socket to the specified server address and port, enabling data exchange between the client and server.

##### Parameters

1. sockfd: The file descriptor of the socket, created using the socket() system call.
2. addr: A pointer to a struct sockaddr containing the server's address and port information. This structure must be cast to (struct sockaddr \*) when passed to connect().
3. addrlen: The size of the address structure (typically obtained using sizeof()).

##### Behaviour

* Returns 0 on successful connection establishment.
* Returns -1 if the connection fails, such as when the server is unreachable or refuses the connection.

##### Example

Here’s an example demonstrating connect() to establish a connection with a server:

c

#include <sys/socket.h>

#include <netinet/in.h>

#include <arpa/inet.h>

#include <stdio.h>

#include <string.h>

int main() {

int sockfd = socket(AF\_INET, SOCK\_STREAM, 0); // Create a TCP socket

if (sockfd < 0) {

perror("Failed to create socket");

return 1;

}

struct sockaddr\_in server\_addr; // Server address structure

memset(&server\_addr, 0, sizeof(server\_addr)); // Initialize to zeros

server\_addr.sin\_family = AF\_INET; // IPv4

server\_addr.sin\_port = htons(8080); // Server port (convert to network byte order)

server\_addr.sin\_addr.s\_addr = inet\_addr("127.0.0.1"); // Server IP address

if (connect(sockfd, (struct sockaddr \*)&server\_addr, sizeof(server\_addr)) < 0) {

perror("Failed to connect to server");

close(sockfd);

return 1;

}

printf("Connected to the server successfully.\n");

// Close the socket when done

close(sockfd);

return 0;

}

##### Output (example):

Connected to the server successfully.

##### Practical Use Cases

1. Client-Server Communication: Establishing connections in backend applications, such as a web client communicating with a web server.
2. Remote Database Access: Connecting to database servers for querying and data retrieval.
3. API Calls: Initiating connections to APIs hosted on remote servers for fetching or sending data.

#### send()

##### Functionality

The send() system call is used to transmit data through a socket that is already connected. It sends data from a buffer to the specified destination, typically used in TCP connections where a reliable communication link has already been established.

##### Parameters

1. sockfd: The file descriptor of the socket, obtained through socket() and connected using connect().
2. buf: A pointer to the buffer containing the data to be sent.
3. len: The number of bytes to send from the buffer.
4. flags: Specifies options that modify the behavior of the send operation (e.g., 0 for default behavior).

##### Behavior

* Returns the number of bytes actually sent, which may be less than len due to network constraints or errors.
* Returns -1 if an error occurs, such as connection failure or insufficient resources.

##### Example

#include <sys/socket.h>

#include <netinet/in.h>

#include <arpa/inet.h>

#include <stdio.h>

#include <string.h>

#include <unistd.h>

int main() {

int sockfd = socket(AF\_INET, SOCK\_STREAM, 0); // Create a TCP socket

if (sockfd < 0) {

perror("Failed to create socket");

return 1;

}

struct sockaddr\_in server\_addr;

memset(&server\_addr, 0, sizeof(server\_addr));

server\_addr.sin\_family = AF\_INET;

server\_addr.sin\_port = htons(8080); // Server port

server\_addr.sin\_addr.s\_addr = inet\_addr("127.0.0.1"); // Server IP

if (connect(sockfd, (struct sockaddr \*)&server\_addr, sizeof(server\_addr)) < 0) {

perror("Failed to connect to server");

close(sockfd);

return 1;

}

const char \*message = "Hello, Server!";

ssize\_t bytesSent = send(sockfd, message, strlen(message), 0); // Send data

if (bytesSent < 0) {

perror("Failed to send data");

} else {

printf("Data sent successfully: %ld bytes\n", bytesSent);

}

close(sockfd);

return 0;

}

##### Output (example):

Data sent successfully: 13 bytes

##### Practical Use Cases

1. Message Transmission: Sending messages or commands from a client to a server, such as in chat applications or REST API requests.
2. Data Upload: Uploading files or data chunks to remote servers in backend systems.
3. Real-Time Applications: Transmitting real-time data in online gaming, streaming, or IoT networks.

#### recv()

##### Functionality

The recv() system call is used to receive data from a connected socket. It reads incoming data from the specified socket and stores it in a buffer for further processing. recv() is primarily used in TCP connections, where the socket is reliably connected to a peer, ensuring that the received data is complete and ordered.

##### Parameters

1. sockfd: The file descriptor for the connected socket, obtained through socket() and connect().
2. buf: A pointer to the buffer where the incoming data will be stored.
3. len: The maximum number of bytes to read into the buffer.
4. flags: Options to modify the behavior of the receive operation (e.g., 0 for default behavior).

##### Behavior

* Returns the number of bytes actually received, which may be less than len if the data is fragmented.
* Returns 0 if the peer has closed the connection.
* Returns -1 if an error occurs, such as connection failure or invalid arguments.

##### Example

#include <sys/socket.h>

#include <netinet/in.h>

#include <arpa/inet.h>

#include <stdio.h>

#include <string.h>

#include <unistd.h>

int main() {

int sockfd = socket(AF\_INET, SOCK\_STREAM, 0); // Create a TCP socket

if (sockfd < 0) {

perror("Failed to create socket");

return 1;

}

struct sockaddr\_in server\_addr;

memset(&server\_addr, 0, sizeof(server\_addr));

server\_addr.sin\_family = AF\_INET;

server\_addr.sin\_port = htons(8080); // Server port

server\_addr.sin\_addr.s\_addr = inet\_addr("127.0.0.1"); // Server IP

if (connect(sockfd, (struct sockaddr \*)&server\_addr, sizeof(server\_addr)) < 0) {

perror("Failed to connect to server");

close(sockfd);

return 1;

}

char buffer[128];

ssize\_t bytesReceived = recv(sockfd, buffer, sizeof(buffer) - 1, 0); // Receive data

if (bytesReceived < 0) {

perror("Failed to receive data");

} else {

buffer[bytesReceived] = '\0'; // Null-terminate the received data

printf("Data received: %s\n", buffer);

}

close(sockfd);

return 0;

}

##### Output (example):

Data received: Welcome to the server!

##### Practical Use Cases

1. Client-Server Communication: Receiving responses from servers after sending requests, such as in HTTP or database queries.
2. Message Handling: Processing incoming messages in chat applications or data streams.
3. Real-Time Data Retrieval: Capturing real-time data in IoT systems, gaming applications, or live feeds.

### 5.System Information Management System Calls

System information management system calls provide processes with the ability to query and retrieve details about the operating system, hardware, and process state. These calls enable developers to access essential system-level information such as process identifiers (PIDs), user IDs (UIDs), system uptime, and resource utilization. They are vital for developing system-monitoring tools, debugging applications, and optimizing resource allocation in backend solutions.

These system calls enhance the ability of programs to dynamically adapt to system conditions, making them a core component of system-level programming and backend management.

#### getpid()

##### Functionality

The getpid() system call retrieves the process ID (PID) of the calling process. A PID is a unique identifier assigned by the operating system to each process running on the system. This system call is commonly used for tracking and managing processes programmatically, especially in debugging or when performing operations on specific processes.

##### Parameters

The getpid() system call does not take any parameters.

##### Behavior

* Returns an integer value representing the PID of the calling process.
* Always succeeds unless an extreme system failure occurs.

##### Example

#include <stdio.h>

#include <unistd.h>

int main() {

pid\_t pid = getpid(); // Get the PID of the calling process

printf("Current Process ID: %d\n", pid);

return 0;

}

##### Output (example):

Current Process ID: 1234

##### Practical Use Cases

1. Process Monitoring: Used to obtain and monitor the PID of running processes in system diagnostics.
2. Inter-Process Communication (IPC): Retrieves the PID to identify the sender or receiver process in IPC mechanisms like signals or shared memory.
3. Debugging: Helps developers identify and track specific processes during debugging or profiling

#### getppid()

##### Functionality

The getppid() system call retrieves the process ID (PID) of the parent process of the calling process. The parent process is the one that initiated or spawned the current process, typically through a mechanism like fork(). This system call is useful for tracking process hierarchies and managing relationships between processes.

##### Parameters

The getppid() system call does not take any parameters.

##### Behavior

* Returns an integer value representing the PID of the parent process.
* Always succeeds unless the process hierarchy is corrupted, which is extremely rare.

##### Example

#include <stdio.h>

#include <unistd.h>

int main() {

pid\_t ppid = getppid(); // Get the PID of the parent process

printf("Parent Process ID: %d\n", ppid);

return 0;

}

##### Output (example):

Parent Process ID: 5678

##### Practical Use Cases

1. Process Hierarchy Tracking: Helps in understanding or debugging the relationship between child and parent processes.
2. Inter-Process Communication (IPC): Identifying the parent process to send or receive signals in IPC mechanisms.
3. Debugging Tools: Useful for creating tools that display or analyze process trees and hierarchies.

#### getuid() and geteuid()

##### Functionality

* getuid(): Retrieves the real user ID (UID) of the calling process. The real UID identifies the user who started the process and is often used for access control.
* geteuid(): Retrieves the effective user ID (EUID) of the calling process. The effective UID determines the permissions the process has, which may differ from the real UID due to mechanisms like setuid (used to temporarily assign elevated privileges).

These system calls are critical for ensuring security and access control in applications, as they allow developers to verify the identity and permissions of running processes.

##### Parameters

Both getuid() and geteuid() do not require any parameters.

##### Behavior

* Returns an integer value representing the UID or EUID of the calling process.
* Always succeeds under normal circumstances.

##### Example

#include <stdio.h>

#include <unistd.h>

int main() {

uid\_t uid = getuid(); // Get the real UID

uid\_t euid = geteuid(); // Get the effective UID

printf("Real User ID: %d\n", uid);

printf("Effective User ID: %d\n", euid);

return 0;

}

##### Output (example):

Real User ID: 1000

Effective User ID: 1000

##### Practical Use Cases

1. Access Control: Verifying the real or effective UID to enforce security policies, such as restricting access to sensitive files or operations.
2. Privilege Management: Detecting and managing elevated privileges in processes using seteuid() and geteuid() mechanisms.
3. Debugging: Useful for troubleshooting permission-related issues in processes.

#### uname()

##### Functionality

The uname() system call retrieves information about the operating system and its kernel. This call is particularly useful for understanding system properties, such as the OS name, version, release date, and hardware architecture. The information is stored in a struct utsname structure, which can be queried for specific details.

##### Parameters

1. utsname: A pointer to a struct utsname, which holds the retrieved system information.

##### Behaviour

* Returns 0 on success, indicating that the system information was retrieved successfully.
* Returns -1 if an error occurs (e.g., due to insufficient permissions or invalid arguments).

##### Example

#include <sys/utsname.h>

#include <stdio.h>

int main() {

struct utsname sysinfo;

if (uname(&sysinfo) == 0) {

printf("System Name: %s\n", sysinfo.sysname);

printf("Node Name: %s\n", sysinfo.nodename);

printf("Release: %s\n", sysinfo.release);

printf("Version: %s\n", sysinfo.version);

printf("Machine: %s\n", sysinfo.machine);

} else {

perror("Failed to retrieve system information");

}

return 0;

}

##### Output (example):

System Name: Linux

Node Name: my-hostname

Release: 5.11.0-41-generic

Version: #45-Ubuntu SMP Wed Oct 20 22:01:10 UTC 2021

Machine: x86\_64

##### Practical Use Cases

1. System Diagnostics: Understanding the kernel version and system architecture for compatibility checks.
2. Monitoring Tools: Collecting system details for debugging or monitoring tools in backend applications.
3. Hardware Optimization: Determining hardware architecture to optimize applications for specific environments

# III. Summary of Findings

#### **Key Takeaways**

1. Process Management System Calls: These calls (exec(), wait(), exit()) are essential for managing processes efficiently. They enable actions like creating and terminating processes, waiting for child processes, and replacing the execution context of processes. These calls ensure seamless execution flow and effective resource allocation in complex applications.
2. File Management System Calls: Calls such as open(), read(), write(), and close() provide robust mechanisms for accessing, manipulating, and managing file data. They are foundational to backend operations involving logging, file-based data storage, and configuration handling.
3. Device Management System Calls: The device-focused calls (read(), write(), ioctl(), select()) allow processes to interact with hardware and monitor device readiness. They abstract hardware intricacies, enabling smooth input/output operations crucial for system-level programming.
4. Network Management System Calls: Network calls (socket(), connect(), send(), recv()) enable processes to communicate over networks. These calls are central to backend services, API integrations, and real-time applications like chat systems and IoT communication.
5. System Information Management System Calls: System-focused calls (getpid(), getppid(), getuid(), geteuid(), uname()) facilitate access to critical details about processes and the operating system. They support monitoring, debugging, and adaptive applications.

# IV. Conclusion

This study highlights how a solid understanding of system calls empowers developers to design efficient, secure, and scalable applications. By leveraging these calls effectively, backend engineers can optimize resource utilization, ensure robust file and device management, build reliable networked applications, and develop sophisticated debugging tools.

Moreover, the exploration of system calls deepens programming knowledge, sharpens problem-solving skills, and fosters innovation. For backend specialists like you, Manshaa, this technical foundation is instrumental in achieving mastery over complex systems and enhancing your ability to contribute impactful solutions to the tech world.

In conclusion, system calls are not just tools—they are the threads that connect applications to the system's core, ensuring harmony and functionality.