Lab 2: System Calls

August 18, 2025

Contents

1 Introduction to System Calls

1.1 Definition and Purpose of System Calls. System calls are the fundamental interface between a process (user program) and the operating system kernel. They are the mechanism through which a user space program requests services from the kernel, such as reading/writing files, creating processes, or communicating with devices.

- **Definition:** System calls allow user programs to invoke functionalities that require privileged CPU operations not accessible in user mode.
- **Purpose:** They provide a safe way to access OS services and hardware without jeopardizing the integrity or security of the system.
- They abstract hardware complexities and enforce resource control and protection policies.

1.2 Distinction Between User Mode and Kernel Mode. Modern CPU architectures implement multiple privilege levels to protect system integrity and maintain security. At a basic level, there are two primary modes of operation:

User Mode:
- **Definition:** User mode is the restricted processing mode in which most application programs execute.
- In this mode, programs have limited access to hardware and cannot directly manipulate critical system resources such as memory management units, I/O devices, or CPU instructions that may affect system stability or security.
- Instructions that could compromise system integrity (e.g., modifying hardware registers or managing memory protection) are disallowed in user mode.
- When a program attempts an operation requiring higher privileges (e.g., reading a file from disk, opening network sockets), it must request the operating system kernel to perform the action on its behalf through a system call.
- User mode restrictions prevent applications from accidentally or maliciously crashing the entire system or interfering with other processes.
- In essence, user mode provides an isolated environment, protecting the operating system and other running programs from faulty or harmful code execution.

Kernel Mode:
- **Definition:** Kernel mode is a privileged mode in which the operating system core (the kernel) executes.
- In this mode, the CPU has unrestricted access to all hardware and system resources, including all processor instructions and memory addresses.
- The kernel can directly interact with hardware devices, manage system memory, schedule tasks, and provide various system services.
- Because kernel mode code runs with full privileges, any bugs or vulnerabilities in the kernel can cause serious system failures or security breaches.
- Thus, operating system kernels are carefully designed, tested, and maintained to preserve system stability and security.
- Kernel mode also handles interrupt processing — reacting immediately to hardware signals like timers, I/O device signals, or exceptions.

CPU Privilege Levels and Protection Rings: Background CPU architectures (such as Intel x86) support multiple privilege levels often called *rings*, ranging from Ring 0 (highest privilege) to Ring 3 (lowest privilege). The operating system kernel runs in Ring 0 (kernel mode), while user applications run in Ring 3 (user mode). The processor enforces these privilege boundaries at hardware level.

Why Two Modes? This separation ensures:

- **Security:** Prevent untrusted user code from damaging critical system resources.
- **Stability:** Faulty or malicious user code cannot crash or halt the entire system.
- **Controlled Access:** All critical operations pass through well-defined system call APIs, allowing auditing, permission checks, and error handling.

Transition Between Modes: User mode programs cannot directly switch to kernel mode; instead, they invoke system calls which cause a controlled, CPU-mediated transition:

1. The program executes a special *trap* or *software interrupt* instruction (e.g., `int 0x80`, `syscall`, or `sysenter` on x86 systems).
2. This instruction saves the program state and transfers control to a predefined interrupt vector in the kernel.
3. The CPU switches to kernel mode privilege level and begins executing the kernel system call handler.
4. The kernel performs requested operations on behalf of the user process.
5. Once finished, the kernel restores program state and switches the CPU context back to user mode, resuming the user program from where it left off.

Because these mode transitions incur overhead (saving/restoring CPU state and switching privilege levels), system calls are considered relatively expensive operations compared to regular function calls. This encourages efficient use of system call APIs.

Example Scenario: When a user program wants to write data to a file:

- It calls the library function `write()` which internally invokes a system call.
- The CPU switches from user mode to kernel mode to execute the kernel code that accesses the file system.
- The kernel performs necessary permission checks and writes data to the disk.
- Control returns to the user program along with the result (number of bytes written or error code).

This careful division between user mode and kernel mode protects the system from errant or malicious software while providing needed services.

In summary, the distinction and controlled transitions between user mode and kernel mode are fundamental to modern operating system security and stability.

1.3 Overview of System Call Execution Flow. The typical flow when a system call is invoked is as follows:

1. **System call invocation:** The user program calls a high-level library function (e.g. `write()`).
2. **Switch from user to kernel mode:** A software interrupt or trap instruction transfers control to a predefined kernel routine.

3. **Parameter validation and permission checking:** The kernel checks arguments and permissions to ensure safe execution.

4. **Service execution:** Kernel executes the requested service (e.g., reading from disk, creating a process).

5. **Return result:** The kernel returns control to user mode, passing back the result or an error code.

This flow ensures controlled access to system resources while maintaining isolation between processes.

2 Categories of System Calls

System calls are grouped into categories based on the type of services they provide to user programs. Understanding these categories will help organize lab topics and provide students with a clearer conceptual framework when learning how these system calls operate. Below is an in-depth discussion of each category with commonly used system calls and their typical purposes.

- **Process Control**
  - `fork()`
    Creates a new process by duplicating the calling process. This child process gets a unique Process ID (PID) but initially inherits a copy of the parent's memory space. This system call is fundamental to process creation in UNIX-like systems.
  - `exec()` family (`execl(), execvp(), execve()`, etc.)
    Replaces the current process image with a new program. Used after `fork()` to run a different executable in the child process.
  - `wait()`
    Causes the parent process to wait until one of its child processes terminates, allowing the parent to retrieve the child's exit status and perform cleanup.
  - `exit()`
    Terminates the calling process and returns an exit status to its parent. Ensures orderly shutdown and resource deallocation.
  - `getpid()`
    Returns the PID of the calling process, useful for process identification and management.
  - **Purpose and Use Cases:**
    Used for process creation, program execution, synchronization between processes, and orderly termination. Many concurrent or multitasking programs rely heavily on these calls to spawn subprocesses and coordinate their operation.
- **File Management**
  - `open()`
    Opens a file or device and returns a file descriptor, which acts as a handle for subsequent operations on that file.
  - `close()`
    Closes an opened file descriptor, releasing the associated resources.
  - `read()`
    Reads bytes of data from an open file descriptor into a user-provided buffer.

- `write()`
  Writes bytes from a user-provided buffer to an open file descriptor.
- `lseek()`
  Repositions the offset (file pointer) within an opened file. Useful for random file access.
- `mkdir()`, `rmdir()`, `unlink()`
  Manage directories and files by creating, removing directories, and deleting files, respectively.
- **Purpose and Use Cases:**
  These calls enable programs to perform low-level input/output operations on files and directories. Essential for file manipulation tasks such as reading configuration files, writing logs, backing up data, or managing filesystem structure.

- **Device Management**
  - `ioctl()`
    Performs device-specific input/output control operations that are not covered by generic calls like `read()` or `write()`. Examples include setting a device parameter or retrieving device status.
  - `read()`, `write()`
    Can also be used on special device files (e.g., `/dev/tty`, `/dev/null`, etc.) to interact with hardware devices or kernel abstractions.
  - **Purpose and Use Cases:**
    Facilitate low-level control and communication with hardware devices or virtual device drivers. Useful in device drivers development and in applications that require hardware manipulation, such as serial port programming or adjusting terminal settings.

- **Information Maintenance**
  - `getpid()`
    Returns the current process identifier.
  - `getuid()` and `setuid()`
    Retrieve or set the user identity of the calling process, governing access permissions and ownership.
  - `time()`
    Returns the current system time, typically the number of seconds elapsed since the epoch (January 1, 1970).
  - **Purpose and Use Cases:**
    These calls provide processes with information about themselves or the system which can control behavior (e.g., permission enforcement) or provide logging and timing information.

- **Inter-Process Communication (IPC)**
  - `pipe()`
    Creates a unidirectional communication channel between two related processes (commonly parent and child).
  - `signal()` and `kill()`
    Signal handling allows processes to asynchronously receive notifications (signals) about events such as interrupts, termination requests, or alarms. `kill()` sends signals to processes.

- Shared memory, message queues, semaphores (via `shmget()`, `msgget()`, `semget()` and related calls)
  Offer more advanced IPC mechanisms that enable multiple processes to communicate and synchronize beyond simple pipes.
- `sockets()`
  Provide endpoints for communication over network protocols, enabling inter-machine communication.
- **Purpose and Use Cases:**
  These system calls enable cooperation and coordination among processes, essential for building complex systems where processes share data or signals. Examples include user-shell communication, client-server applications, and synchronization in concurrent programming.

- **Protection and Security**
  - During system calls, the kernel performs *permission checks* to verify that the calling process has adequate rights (read, write, execute) to carry out the requested operation.
  - Access control is enforced based on user credentials, ownerships, and file permission bits.
  - Attempts to violate security policies (such as opening a file without read permission) cause the system call to fail, usually returning an error code like `EACCES`.
  - Some system calls are specifically aimed at security and access control, e.g., `setuid()` to change user privileges.
  - **Purpose and Use Cases:**
    Protection mechanisms prevent unauthorized access or modification to resources, preserving system integrity and confidentiality. Teaching these concepts leads students to understand OS security fundamentals and error handling.

3 Essential System Calls and Hands-on Demonstrations

This section presents detailed descriptions and practical demonstrations of key system calls categorized by their common uses: process creation and management, file and directory operations, error handling mechanisms, and inter-process communication (IPC). Each subsection explains the purpose and usage of respective system calls, accompanied by well-commented C code snippets to facilitate hands-on experimentation and in-depth understanding.

3.1 Process Creation and Management. Process management system calls enable creation, execution control, synchronization, and termination of processes in a multitasking operating system. These calls form the foundation of concurrent and parallel program execution.

`fork()`: • Creates a new process by duplicating the calling (parent) process.
- Returns `0` in the child process, and the child's Process ID (PID) in the parent process.
- Returns `-1` if the fork fails (e.g., due to system resource exhaustion).
- Both parent and child continue execution independently from the point after `fork()`.
- Commonly used for spawning new processes that can execute concurrently.

`exec()` **family:** • Replaces the current process memory image with a new program.
- Variants include `execl()`, `execv()`, `execvp()`, etc., differing in argument passing style.
- Successful call does not return; the new program starts executing.

- Typically called after `fork()` in the child process to execute a different program.

`wait()` and `waitpid()`:
- Causes the parent process to block until one or more child processes terminate.
- Retrieves termination status and resource usage information of the child.
- Enables synchronization and clean process termination handling.

`exit()`:
- Terminates the calling process and returns an exit status to its parent process.
- Releases all resources held by the process.
- Exit status can be retrieved by parent via `wait()`.

`getpid()`:
- Retrieves the numeric PID of the current process.
- Useful for process identification, logging, and debugging.

Example program demonstrating `fork()` and `getpid()` system calls:

```
#include <stdio.h>      // For printf, perror
#include <stdlib.h>     // For exit, EXIT_FAILURE
#include <unistd.h>     // For fork, getpid, getppid
#include <sys/types.h>  // For pid_t

int main() {
    // Create a new process
    pid_t pid = fork();

    // Check if the fork() call failed
    if (pid == -1) {
        perror("fork"); // Print the system error message
        exit(EXIT_FAILURE);
    }

    // This block is executed by the child process
    if (pid == 0) {
        printf("Child process:  PID = %d, Parent PID = %d\n", getpid(), getppid());
        // The child can perform different tasks or exec a new program here
    }
    // This block is executed by the parent process
    else {
        printf("Parent process: PID = %d, Child PID = %d\n", getpid(), pid);
        // The parent can wait for the child or continue its execution
    }

    return 0;
}
```

Note: This program creates a child process and both parent and child print their process identifiers, illustrating how `fork()` works.

3.2 File and Directory Operations. File management system calls allow programs to perform low-level operations on files and directories, such as opening, reading, writing, and manipulating filesystem structure. These are essential for persistent storage interaction.

`open()`: • Opens a file specified by pathname with access flags (e.g., `O_RDONLY`, `O_WRONLY`, `O_RDWR`).

- Returns a non-negative file descriptor on success, or `-1` on failure.
- Can specify additional flags for behaviors like `O_CREAT` (create if doesn't exist) and file permission modes.

`close()`: • Closes an open file descriptor, freeing associated system resources.

- Failing to close can lead to resource leaks.

`read()`: • Reads up to a specified number of bytes from a file descriptor into a buffer.

- Returns number of bytes actually read or `0` if EOF is reached.
- Returns `-1` and sets `errno` on failure.

`write()`: • Writes data from buffer to the file descriptor.

- Returns number of bytes written or `-1` on error.
- Important to check that all intended bytes are written.

`lseek()`: • Repositions the offset (file pointer) of an open file descriptor.

- Allows random access reads/writes.
- Parameters specify offset and reference point (`SEEK_SET`, `SEEK_CUR`, `SEEK_END`).

`mkdir()`, `rmdir()`, `unlink()`: • `mkdir()` creates a new directory with specified permissions.

- `rmdir()` removes an empty directory.
- `unlink()` deletes a file from the filesystem.

Example: A robust program to copy contents from a source file to a destination file using low-level system calls:

```
#include <stdio.h>
#include <fcntl.h>
#include <unistd.h>
#include <errno.h>
#include <string.h>

#define BUFFER_SIZE 1024

int main(int argc, char *argv[]) {
    if (argc != 3) {
        fprintf(stderr, "Usage: %s <source> <destination>\n", argv[0]);
        return 1;
    }

    // Open source file for reading
    int source_fd = open(argv[1], O_RDONLY);
```

```
    if (source_fd < 0) {
        perror("Error opening source file");
        return 1;
    }

    // Open/create destination file for writing (truncate if exists)
    int dest_fd = open(argv[2], O_WRONLY | O_CREAT | O_TRUNC, 0644);
    if (dest_fd < 0) {
        perror("Error opening destination file");
        close(source_fd);
        return 1;
    }

    char buffer[BUFFER_SIZE];
    ssize_t bytes_read;

    // Loop reading from source and writing to destination
    while ((bytes_read = read(source_fd, buffer, sizeof(buffer))) > 0) {
        ssize_t bytes_written = 0;
        while (bytes_written < bytes_read) {
            ssize_t result = write(dest_fd, buffer + bytes_written, bytes_read - bytes_wri
            if (result < 0) {
                perror("Error writing to destination file");
                close(source_fd);
                close(dest_fd);
                return 1;
            }
            bytes_written += result;
        }
    }

    if (bytes_read < 0) {
        perror("Error reading from source file");
    }

    close(source_fd);
    close(dest_fd);

    return (bytes_read < 0) ? 1 : 0;
}
```

This program:

- Validates command line arguments.
- Opens files using `open()`.
- Performs full read-write loops handling partial writes.
- Catches and reports errors using `perror()`.

- Closes open file descriptors properly to avoid resource leaks.

3.3 Error Handling: Return Values and `errno` Usage. Robust programs must always check for errors on system calls because many issues can occur, such as resource exhaustion, permissions problems, or hardware failures.

- Most system calls return `-1` upon encountering an error.
- The global variable `errno` is set by the system to indicate the specific error cause.
- Use `perror()` to print a descriptive message associated with `errno`.
- Alternatively, use `strerror(errno)` to retrieve an error string programmatically.
- Always check return values to handle errors gracefully, avoiding undefined behaviors or crashes.

Example program illustrating error handling when attempting to open a non-existent file:

```
#include <fcntl.h>
#include <stdio.h>
#include <errno.h>

int main() {
    int fd = open("nonexistent.txt", O_RDONLY);
    if (fd == -1) {
        // Print descriptive error message to stderr
        perror("Failed to open file");
        return 1;
    }

    // If open succeeded, close file descriptor
    close(fd);
    return 0;
}
```

3.4 Inter-Process Communication (IPC). IPC system calls enable processes to communicate and synchronize their actions. This is critical for cooperative multitasking and client-server architectures.

`pipe()`:
- Creates a unidirectional communication channel represented by two file descriptors: one for reading and one for writing.
- Typically used between a parent and child process created with `fork()`.
- Data written to the write-end can be read from the read-end in a FIFO manner.

`dup()` and `dup2()`:
- Duplicate an existing file descriptor to create a new descriptor pointing to the same file/table entry.
- Useful for redirecting inputs/outputs, e.g., redirecting standard output to a file or pipe.

`kill()`: • Sends a specified signal to the target process, which could request termination, stop, or restart, or invoke a signal handler.

`signal()`: • Registers a handler function to asynchronously catch and handle various signals like `SIGINT` (interrupt) or `SIGTERM` (termination).

Example demonstrating pipe communication between parent and child processes:

```
#include <stdio.h>
#include <unistd.h>
#include <string.h>
#include <sys/types.h>
#include <stdlib.h>
#include <sys/wait.h>

int main() {
    int pipefd[2];
    pid_t pid;
    char buffer[100];

    // Create the pipe
    if (pipe(pipefd) == -1) {
        perror("pipe");
        exit(EXIT_FAILURE);
    }

    // Fork the process
    pid = fork();
    if (pid == -1) {
        perror("fork");
        exit(EXIT_FAILURE);
    }

    if (pid == 0) {
        // Child process: read from pipe
        close(pipefd[1]); // Close unused write end

        ssize_t count = read(pipefd[0], buffer, sizeof(buffer) - 1);
        if (count == -1) {
            perror("read");
            exit(EXIT_FAILURE);
        }
        buffer[count] = '\0';  // Null-terminate string

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

        close(pipefd[0]);
        exit(EXIT_SUCCESS);
    } else {
        // Parent process: write to pipe
```

```
        close(pipefd[0]); // Close unused read end

        const char *message = "Hello from the parent process!";
        ssize_t len = strlen(message);

        if (write(pipefd[1], message, len) != len) {
            perror("write");
            exit(EXIT_FAILURE);
        }

        close(pipefd[1]);

        // Optionally wait for child to finish here
        wait(NULL);
    }

    return 0;
}
```

This example illustrates:

- Creating a pipe descriptor pair before `fork()`.
- Closing unused ends of the pipe in each process to avoid deadlocks.
- The parent writing a string message into the pipe.
- The child reading and printing the message from the pipe.

4 Essential System Calls and Hands-on Demonstrations

This section presents detailed descriptions and practical demonstrations of key system calls categorized by their common uses: process creation and management, file and directory operations, error handling mechanisms, and inter-process communication (IPC). Each subsection explains the purpose and usage of respective system calls, accompanied by well-commented C code snippets to facilitate hands-on experimentation and in-depth understanding.

4.1 Process Creation and Management. Process management system calls enable creation, execution control, synchronization, and termination of processes in a multitasking operating system. These calls form the foundation of concurrent and parallel program execution.

`fork()`: • Creates a new process by duplicating the calling (parent) process.
- Returns `0` in the child process, and the child's Process ID (PID) in the parent process.
- Returns `-1` if the fork fails (e.g., due to system resource exhaustion).
- Both parent and child continue execution independently from the point after `fork()`.
- Commonly used for spawning new processes that can execute concurrently.

`exec()` **family:** • Replaces the current process memory image with a new program.
- Variants include `execl()`, `execv()`, `execvp()`, etc., differing in argument passing style.

- Successful call does not return; the new program starts executing.
- Typically called after `fork()` in the child process to execute a different program.

`wait()` and `waitpid()`:
- Causes the parent process to block until one or more child processes terminate.
- Retrieves termination status and resource usage information of the child.
- Enables synchronization and clean process termination handling.

`exit()`:
- Terminates the calling process and returns an exit status to its parent process.
- Releases all resources held by the process.
- Exit status can be retrieved by parent via `wait()`.

`getpid()`:
- Retrieves the numeric PID of the current process.
- Useful for process identification, logging, and debugging.

Example program demonstrating `fork()` and `getpid()` system calls:

```
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
#include <stdlib.h>

int main() {
    pid_t pid = fork();

    if (pid == -1) {
        // Fork failed
        perror("fork");
        exit(EXIT_FAILURE);
    }

    if (pid == 0) {
        // Child process
        printf("Child process: PID = %d, Parent PID = %d\n", getpid(), getppid());
        // Child can perform different tasks or exec a new program here
    } else {
        // Parent process
        printf("Parent process: PID = %d, Child PID = %d\n", getpid(), pid);
        // Parent can wait for child or continue execution
    }

    return 0;
}
```

Note: This program creates a child process and both parent and child print their process identifiers, illustrating how `fork()` works.

4.2 File and Directory Operations. File management system calls allow programs to perform low-level operations on files and directories, such as opening, reading, writing, and manipulating filesystem structure. These are essential for persistent storage interaction.

`open()`: • Opens a file specified by pathname with access flags (e.g., `O_RDONLY`, `O_WRONLY`, `O_RDWR`).

- Returns a non-negative file descriptor on success, or `-1` on failure.
- Can specify additional flags for behaviors like `O_CREAT` (create if doesn't exist) and file permission modes.

`close()`: • Closes an open file descriptor, freeing associated system resources.

- Failing to close can lead to resource leaks.

`read()`: • Reads up to a specified number of bytes from a file descriptor into a buffer.

- Returns number of bytes actually read or `0` if EOF is reached.
- Returns `-1` and sets `errno` on failure.

`write()`: • Writes data from buffer to the file descriptor.

- Returns number of bytes written or `-1` on error.
- Important to check that all intended bytes are written.

`lseek()`: • Repositions the offset (file pointer) of an open file descriptor.

- Allows random access reads/writes.
- Parameters specify offset and reference point (`SEEK_SET`, `SEEK_CUR`, `SEEK_END`).

`mkdir()`, `rmdir()`, `unlink()`: • `mkdir()` creates a new directory with specified permissions.

- `rmdir()` removes an empty directory.
- `unlink()` deletes a file from the filesystem.

Example: A robust program to copy contents from a source file to a destination file using low-level system calls:

```
#include <stdio.h>
#include <fcntl.h>
#include <unistd.h>
#include <errno.h>
#include <string.h>

#define BUFFER_SIZE 1024

int main(int argc, char *argv[]) {
    if (argc != 3) {
        fprintf(stderr, "Usage: %s <source> <destination>\n", argv[0]);
        return 1;
    }

    // Open source file for reading
    int source_fd = open(argv[1], O_RDONLY);
```

```
    if (source_fd < 0) {
        perror("Error opening source file");
        return 1;
    }

    // Open/create destination file for writing (truncate if exists)
    int dest_fd = open(argv[2], O_WRONLY | O_CREAT | O_TRUNC, 0644);
    if (dest_fd < 0) {
        perror("Error opening destination file");
        close(source_fd);
        return 1;
    }

    char buffer[BUFFER_SIZE];
    ssize_t bytes_read;

    // Loop reading from source and writing to destination
    while ((bytes_read = read(source_fd, buffer, sizeof(buffer))) > 0) {
        ssize_t bytes_written = 0;
        while (bytes_written < bytes_read) {
            ssize_t result = write(dest_fd, buffer + bytes_written, bytes_read - bytes_wri
            if (result < 0) {
                perror("Error writing to destination file");
                close(source_fd);
                close(dest_fd);
                return 1;
            }
            bytes_written += result;
        }
    }

    if (bytes_read < 0) {
        perror("Error reading from source file");
    }

    close(source_fd);
    close(dest_fd);

    return (bytes_read < 0) ? 1 : 0;
}
```

This program:

- Validates command line arguments.
- Opens files using `open()`.
- Performs full read-write loops handling partial writes.
- Catches and reports errors using `perror()`.

- Closes open file descriptors properly to avoid resource leaks.

4.3 Error Handling: Return Values and `errno` Usage. Robust programs must always check for errors on system calls because many issues can occur, such as resource exhaustion, permissions problems, or hardware failures.

- Most system calls return `-1` upon encountering an error.
- The global variable `errno` is set by the system to indicate the specific error cause.
- Use `perror()` to print a descriptive message associated with `errno`.
- Alternatively, use `strerror(errno)` to retrieve an error string programmatically.
- Always check return values to handle errors gracefully, avoiding undefined behaviors or crashes.

Example program illustrating error handling when attempting to open a non-existent file:

```
#include <fcntl.h>
#include <stdio.h>
#include <errno.h>

int main() {
    int fd = open("nonexistent.txt", O_RDONLY);
    if (fd == -1) {
        // Print descriptive error message to stderr
        perror("Failed to open file");
        return 1;
    }

    // If open succeeded, close file descriptor
    close(fd);
    return 0;
}
```

4.4 Inter-Process Communication (IPC). IPC system calls enable processes to communicate and synchronize their actions. This is critical for cooperative multitasking and client-server architectures.

`pipe()`:
- Creates a unidirectional communication channel represented by two file descriptors: one for reading and one for writing.
- Typically used between a parent and child process created with `fork()`.
- Data written to the write-end can be read from the read-end in a FIFO manner.

`dup()` and `dup2()`:
- Duplicate an existing file descriptor to create a new descriptor pointing to the same file/table entry.
- Useful for redirecting inputs/outputs, e.g., redirecting standard output to a file or pipe.

`kill()`: • Sends a specified signal to the target process, which could request termination, stop, or restart, or invoke a signal handler.

`signal()`: • Registers a handler function to asynchronously catch and handle various signals like `SIGINT` (interrupt) or `SIGTERM` (termination).

Example demonstrating pipe communication between parent and child processes:

```
#include <stdio.h>
#include <unistd.h>
#include <string.h>
#include <sys/types.h>
#include <stdlib.h>
#include <sys/wait.h>

int main() {
    int pipefd[2];
    pid_t pid;
    char buffer[100];

    // Create the pipe
    if (pipe(pipefd) == -1) {
        perror("pipe");
        exit(EXIT_FAILURE);
    }

    // Fork the process
    pid = fork();
    if (pid == -1) {
        perror("fork");
        exit(EXIT_FAILURE);
    }

    if (pid == 0) {
        // Child process: read from pipe
        close(pipefd[1]); // Close unused write end

        ssize_t count = read(pipefd[0], buffer, sizeof(buffer) - 1);
        if (count == -1) {
            perror("read");
            exit(EXIT_FAILURE);
        }
        buffer[count] = '\0';  // Null-terminate string

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

        close(pipefd[0]);
        exit(EXIT_SUCCESS);
    } else {
        // Parent process: write to pipe
```

```
        close(pipefd[0]); // Close unused read end

        const char *message = "Hello from the parent process!";
        ssize_t len = strlen(message);

        if (write(pipefd[1], message, len) != len) {
            perror("write");
            exit(EXIT_FAILURE);
        }

        close(pipefd[1]);

        // Optionally wait for child to finish here
        wait(NULL);
    }

    return 0;
}
```

This example illustrates:

- Creating a pipe descriptor pair before `fork()`.
- Closing unused ends of the pipe in each process to avoid deadlocks.
- The parent writing a string message into the pipe.
- The child reading and printing the message from the pipe.

5 Appendix: Detailed System Call Reference Table

System Call	Synopsis and Parameters	Description and Notes
`fork()`	pid_t fork(void);	Creates a child process; returns 0 in child, PID in parent, -1 on failure.
`execvp()`	int execvp(const char *file, char *const argv[]);	Replaces process image with a new program. Does not return on success.
`wait()`	pid_t wait(int *status);	Waits for any child process to terminate; returns PID or -1 on error.
`exit()`	void exit(int status);	Terminates calling process and returns status to parent.
`getpid()`	pid_t getpid(void);	Returns calling process's PID.
`open()`	int open(const char *pathname, int flags, ...);	Opens a file, returns file descriptor or -1 on error.
`close()`	int close(int fd);	Closes file descriptor; returns 0 on success, -1 on error.
`read()`	ssize_t read(int fd, void *buf, size_t count);	Reads bytes from fd to buf; returns bytes read or -1 on error.
`write()`	ssize_t write(int fd, const void *buf, size_t count);	Writes bytes from buf to fd; returns bytes written or -1 on error.
`lseek()`	off_t lseek(int fd, off_t offset, int whence);	Moves file pointer; returns new offset or -1 on error.
`pipe()`	int pipe(int pipefd[2]);	Creates IPC pipe; pipefd[0] for read, pipefd[1] for write. Returns 0 or -1.
`dup()`	int dup(int oldfd);	Duplicates fd; returns new fd or -1 on error.
`kill()`	int kill(pid_t pid, int sig);	Sends signal sig to PID pid; returns 0 or -1 on error.
`signal()`	void (*signal(int sig, void (*func)(int)))(int);	Sets signal handler for sig; returns previous handler or SIG_{ERR}.

6 Windows and Linux System Call Equivalents

Table 2: Comparison of Windows and Linux System Calls

	Windows	**Linux**
Process Control	CreateProcess()	fork()
	ExitProcess()	exit()
	WaitForSingleObject()	wait()
File Manipulation	CreateFile()	open()
	ReadFile()	read()
	WriteFile()	write()
	CloseHandle()	close()
Device Manipulation	SetConsoleMode()	ioctl()
	ReadConsole()	read()
	WriteConsole()	write()
Information Maintenance	GetCurrentProcessID()	getpid()
	SetTimer()	alarm()
	Sleep()	sleep()
Communication	CreatePipe()	pipe()
	CreateFileMapping()	shmget()
	MapViewOfFile()	mmap()
Protection	SetFileSecurity()	chmod()
	InitializeSecurityDescriptor()	chown()
	SetSecurityDescriptorGroup()	umask()

Assignment Question

Exploring Fundamental System Calls in the Linux Operating System

Objectives:.

1. **Write a C program that:**
   - Creates a child process using `fork()`.
   - The child process uses `execvp()` to run a different executable.
   - The parent process waits for the child to finish using `wait()` and prints the child's exit status.
2. **Implement a file copy utility in C that:**
   - Copies contents from a source file to a destination file using `open()`, `read()`, `write()`, and `close()`.
   - Performs thorough error checking and handles files of any size efficiently.
3. **Develop a program that:**
   - Creates a pipe before `fork()`.
   - The parent writes a message into the pipe.
   - The child reads the message and prints it.
   - Both processes close unused pipe ends properly.
4. **For all programs:**
   - Include robust error handling using `perror()` or equivalent.
   - Ensure proper resource cleanup on errors.