**System Call**

System calls are a critical interface between an application and the operating system kernel. They provide a means for user-space applications to request services and resources managed by the kernel, such as file operations, process control, networking, and inter-process communication. System calls act as a gateway for executing privileged operations, ensuring security and stability within the system.

**Detailed Explanation of System Calls:**

* **Purpose and Role:**
* System calls allow user-space programs to perform operations that require higher privileges, such as hardware access or manipulating process states.
* They abstract complex hardware and low-level operations into a simpler, consistent API that applications can use.

System calls serve several purposes:

* **Accessing Kernel Services**: User applications need to perform operations that require privileged access. System calls provide a controlled interface for accessing these services.
* **Abstracting Hardware Details**: They abstract the hardware details and provide a consistent API, making it easier for developers to write applications.
* **Ensuring Security and Stability**: Direct hardware access by user applications can compromise system stability and security. System calls ensure that all operations are mediated and validated by the kernel.
* **Mechanism:**

When a system call is invoked, the following steps typically occur:

* **Trap Instruction**: The user program executes a special instruction (a trap or syscall) that switches the CPU to kernel mode.
* **System Call Handler**: The kernel handles the system call, performing the requested operation.
* **Return to User Mode**: The kernel switches back to user mode and returns the result of the system call to the user program with any necessary results or error codes.
* **Common System Calls:**
* File Operations: open, read, write, close
* Process Control: fork, exec, wait, exit
* Memory Management: mmap, munmap
* Networking: socket, bind, listen, accept
* Miscellaneous: getpid, kill, time

Example Code Demonstrating System Calls in C

Here's a simple example in C that demonstrates the use of some basic system calls to create a new process, open a file, and read its contents.

**Example 1:**

#include <stdio.h>

#include <stdlib.h>

#include <fcntl.h>

#include <unistd.h>

#include <sys/types.h>

#include <sys/wait.h>

#define BUFFER\_SIZE 256

int main() {

pid\_t pid;

int fd;

char buffer[BUFFER\_SIZE];

ssize\_t bytesRead;

**// Fork a new process**

pid = fork();

if (pid < 0) {

perror("fork");

exit(EXIT\_FAILURE);

}

if (pid == 0) {

**// Child process**

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

**// Open a file for reading**

fd = open("example.txt", O\_RDONLY);

if (fd < 0) {

perror("open");

exit(EXIT\_FAILURE);

}

**// Read contents of the file**

while ((bytesRead = read(fd, buffer, BUFFER\_SIZE)) > 0) {

write(STDOUT\_FILENO, buffer, bytesRead);

}

if (bytesRead < 0) {

perror("read");

close(fd);

exit(EXIT\_FAILURE);

}

**// Close the file**

close(fd);

exit(EXIT\_SUCCESS);

} else {

**// Parent process**

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

**// Wait for the child process to complete**

wait(NULL);

}

return 0;

}

**Explanation:**

* **Forking a Process (fork):**
* The fork system call creates a new process by duplicating the existing process. The new process (child) runs concurrently with the parent process.
* pid will be zero in the child process, and the child's PID in the parent process. A negative value indicates an error.
* **Opening a File (open):**
* The open system call opens a file descriptor for the specified file (example.txt in this case) with read-only access (O\_RDONLY).
* If the file cannot be opened, open returns -1, and perror prints an error message.
* **Reading from a File (read):**
* The read system call reads up to BUFFER\_SIZE bytes from the file descriptor fd into the buffer.
* The return value indicates the number of bytes read, or -1 if an error occurs.
* **Writing to Standard Output (write):**
* The write system call writes the contents of the buffer to the standard output (STDOUT\_FILENO).
* **Closing a File (close):**
* The close system call closes the file descriptor, releasing the resources.
* **Waiting for a Child Process (wait):**
* The wait system call makes the parent process wait until all its child processes have terminated. It blocks the parent until the child completes, ensuring proper synchronization.

This example demonstrates fundamental system calls for process creation, file handling, and I/O operations, showcasing how user-space applications interact with the kernel to perform essential tasks.

**Example 2:**

**Example 2.1: File Operations**

#include <fcntl.h> // For open()

#include <unistd.h> // For read(), write(), close()

#include <stdio.h> // For perror()

int main() {

int fd;

char buffer[100];

ssize\_t bytesRead;

**// Open a file for reading**

fd = open("example.txt", O\_RDONLY);

if (fd == -1) {

perror("open");

return 1;

}

**// Read from the file**

bytesRead = read(fd, buffer, sizeof(buffer) - 1);

if (bytesRead == -1) {

perror("read");

close(fd);

return 1;

}

**// Null-terminate the buffer and print the contents**

buffer[bytesRead] = '\0';

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

**// Close the file**

if (close(fd) == -1) {

perror("close");

return 1;

}

return 0;

}

In this example:

* open() is used to open a file and obtain a file descriptor.
* read() reads data from the file into a buffer.
* close() closes the file descriptor.

**Example 2.2: Process Control**

#include <stdio.h>

#include <unistd.h>

#include <sys/types.h>

#include <sys/wait.h>

int main() {

pid\_t pid;

int status;

**// Fork a new process**

pid = fork();

if (pid == -1) {

**// Fork failed**

perror("fork");

return 1;

} else if (pid == 0) {

**// Child process**

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

**// Replace the current process image with a new one (exec)**

execl("/bin/ls", "ls", "-l", (char \*)NULL);

**// If execl() returns, it must have failed**

perror("execl");

return 1;

} else {

**// Parent process**

printf("Parent process (PID: %d), waiting for child (PID: %d)\n", getpid(), pid);

**// Wait for the child process to finish**

if (waitpid(pid, &status, 0) == -1) {

perror("waitpid");

return 1;

}

**// Check if the child process terminated normally**

if (WIFEXITED(status)) {

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

} else {

printf("Child did not exit normally\n");

}

}

return 0;

}

In this example:

* fork() creates a new process. The child process gets a process ID (PID) of 0, while the parent gets the PID of the child.
* In the child process, execl() replaces the current process image with the ls command.
* The parent process uses waitpid() to wait for the child process to finish.
* WIFEXITED() and WEXITSTATUS() macros are used to check the termination status of the child process.

**Example 2.3: Memory Management**

#include <stdio.h>

#include <stdlib.h>

#include <sys/mman.h>

#include <string.h>

#include <unistd.h>

int main() {

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

size\_t length = strlen(message) + 1;

**// Allocate memory using mmap**

char \*addr = mmap(NULL, length, PROT\_READ | PROT\_WRITE, MAP\_ANONYMOUS | MAP\_PRIVATE, -1, 0);

if (addr == MAP\_FAILED) {

perror("mmap");

return 1;

}

**// Write to the allocated memory**

strcpy(addr, message);

printf("Mapped memory contains: %s\n", addr);

**// Deallocate memory using munmap**

if (munmap(addr, length) == -1) {

perror("munmap");

return 1;

}

return 0;

}

In this example:

* mmap() allocates a memory region that is both readable and writable.
* The allocated memory is used to store a string.
* munmap() deallocates the memory region.

**Example 2.4: Inter-Process Communication (Pipe)**

#include <stdio.h>

#include <unistd.h>

int main() {

int pipefd[2];

pid\_t pid;

char buffer[30];

**// Create a pipe**

if (pipe(pipefd) == -1) {

perror("pipe");

return 1;

}

**// Fork a new process**

pid = fork();

if (pid == -1) {

perror("fork");

return 1;

} else if (pid == 0) {

**// Child process**

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

write(pipefd[1], "Hello from child", 16);

close(pipefd[1]); **// Close write end**

} else {

**// Parent process**

close(pipefd[1]); **// Close unused write end**

read(pipefd[0], buffer, sizeof(buffer));

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

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

}

return 0;

}

In this example:

* pipe() creates a unidirectional data channel.
* The child process writes a message to the pipe.
* The parent process reads the message from the pipe.

**Example 3:**

#include <stdio.h>

#include <stdlib.h>

#include <unistd.h>

#include <fcntl.h>

#include <sys/wait.h>

#include <sys/types.h>

#include <sys/stat.h>

#include <string.h>

#define BUFFER\_SIZE 256

int main() {

pid\_t pid;

int fd;

char buffer[BUFFER\_SIZE];

ssize\_t bytesRead;

**// Create a named pipe (FIFO)**

const char \*fifoPath = "/tmp/myfifo";

if (mkfifo(fifoPath, 0666) == -1) {

perror("mkfifo");

exit(EXIT\_FAILURE);

}

**// Fork a new process**

pid = fork();

if (pid < 0) {

perror("fork");

exit(EXIT\_FAILURE);

}

if (pid == 0) {

**// Child process**

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

**// Open the named pipe for writing**

fd = open(fifoPath, O\_WRONLY);

if (fd < 0) {

perror("open");

exit(EXIT\_FAILURE);

}

**// Write a message to the named pipe**

const char \*message = "Hello from child process!\n";

if (write(fd, message, strlen(message)) == -1) {

perror("write");

close(fd);

exit(EXIT\_FAILURE);

}

**// Close the named pipe**

close(fd);

exit(EXIT\_SUCCESS);

} else {

**// Parent process**

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

// Open the named pipe for reading

fd = open(fifoPath, O\_RDONLY);

if (fd < 0) {

perror("open");

exit(EXIT\_FAILURE);

}

**// Read from the named pipe**

bytesRead = read(fd, buffer, BUFFER\_SIZE);

if (bytesRead == -1) {

perror("read");

close(fd);

exit(EXIT\_FAILURE);

}

**// Null-terminate the string and print it**

buffer[bytesRead] = '\0';

printf("Received from child: %s", buffer);

**// Close the named pipe**

close(fd);

**// Wait for the child process to complete**

wait(NULL);

**// Remove the named pipe**

if (unlink(fifoPath) == -1) {

perror("unlink");

exit(EXIT\_FAILURE);

}

printf("Named pipe removed and parent process exiting.\n");

}

return 0;

**Explanation:**

* **Creating a Named Pipe (`mkfifo`):**
* The `mkfifo` system call creates a named pipe (FIFO) at the specified path (`/tmp/myfifo`) with read/write permissions (0666). Named pipes provide a way for unrelated processes to communicate.
* **Forking a Process (`fork`):**
* The `fork` system call creates a new process by duplicating the existing process. The new process is known as the child process.
* In the parent process, `fork` returns the PID of the child process. In the child process, `fork` returns 0.
* **Child Process Operations:**
* The child process opens the named pipe for writing using the `open` system call with the `O\_WRONLY` flag.
* It writes a message to the pipe using the `write` system call.
* After writing the message, the child process closes the named pipe using the `close` system call and exits with `exit(EXIT\_SUCCESS)`.
* **Parent Process Operations:**
* The parent process opens the named pipe for reading using the `open` system call with the `O\_RDONLY` flag.
* It reads the message from the pipe using the `read` system call and stores the result in the buffer.
* The parent process then null-terminates the buffer and prints the received message.
* After reading, the parent process closes the named pipe using the `close` system call.
* The parent process waits for the child process to complete using the `wait` system call, ensuring synchronization.
* Finally, the parent process removes the named pipe using the `unlink` system call.

This example demonstrates several key systems calls in Linux, showcasing process creation, file operations, and inter-process communication. Each system call plays a vital role in enabling the application to interact with the operating system kernel to perform necessary tasks securely and efficiently.