**LAB ASSIGNMENT-04**

Aim: 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().

**Purpose:**

Creates a new process by duplicating the calling (parent) process. The new process is called the child process.

**Syntax:**

c

CopyEdit

pid\_t fork(void);

**Return Value:**

* 0 to the child process
* Child’s PID to the parent process
* -1 on failure

**Example:**

c

CopyEdit

#include <stdio.h>

#include <unistd.h>

int main() {

pid\_t pid = fork();

if (pid == 0) {

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

} else if (pid > 0) {

printf("Parent Process: PID = %d, Child PID = %d\n", getpid(), pid);

} else {

perror("fork failed");

}

return 0;

}

**2. exec()**

**Purpose:**

Replaces the current process image with a new program. This means the current process memory is replaced, and the new program starts executing.

**Syntax (example variant execl):**

c

CopyEdit

int execl(const char \*path, const char \*arg0, ..., NULL);

**Example:**

c

CopyEdit

#include <unistd.h>

int main() {

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

return 0; // This line is not reached if execl is successful

}

**3. wait()**

**Purpose:**

Makes the parent process wait until its child process terminates.

**Syntax:**

c

CopyEdit

pid\_t wait(int \*status);

**Example:**

c

CopyEdit

#include <stdio.h>

#include <sys/wait.h>

#include <unistd.h>

int main() {

pid\_t pid = fork();

if (pid == 0) {

printf("Child is running...\n");

} else {

wait(NULL); // Wait for the child to finish

printf("Parent: Child finished.\n");

}

return 0;

}

**4. exit()**

**Purpose:**

Terminates the current process and returns a status code to the parent.

**Syntax:**

c

CopyEdit

void exit(int status);

**Example:**

c

CopyEdit

#include <stdlib.h>

int main() {

printf("Process is exiting...\n");

exit(0);

}

These four system calls—fork(), exec(), wait(), and exit()—form the foundation of process management in Linux.

* fork() enables multitasking by creating new processes.
* exec() loads and executes a new program.
* wait() ensures process synchronization.
* exit() cleanly terminates processes.

2. File Management System calls

open(), read(), write(), close().

**1. open()**

**Purpose:**

Opens a file and returns a file descriptor.

**Syntax:**

c

CopyEdit

int open(const char \*pathname, int flags, mode\_t mode);

* pathname: Name of the file.
* flags: File access modes (O\_RDONLY, O\_WRONLY, O\_RDWR, etc.).
* mode: File permission bits (used when creating a file).

**Example:**

c

CopyEdit

#include <fcntl.h>

#include <stdio.h>

int main() {

int fd = open("example.txt", O\_CREAT | O\_WRONLY, 0644);

if (fd < 0) {

perror("open");

return 1;

}

printf("File opened with file descriptor: %d\n", fd);

return 0;

}

**2. read()**

**Purpose:**

Reads data from a file descriptor into a buffer.

**Syntax:**

c

CopyEdit

ssize\_t read(int fd, void \*buf, size\_t count);

* fd: File descriptor.
* buf: Pointer to a buffer.
* count: Number of bytes to read.

**Example:**

c

CopyEdit

#include <fcntl.h>

#include <unistd.h>

#include <stdio.h>

int main() {

char buffer[100];

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

if (fd < 0) {

perror("open");

return 1;

}

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

buffer[bytesRead] = '\0';

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

close(fd);

return 0;

}

**3. write()**

**Purpose:**

Writes data from a buffer to a file descriptor.

**Syntax:**

c

CopyEdit

ssize\_t write(int fd, const void \*buf, size\_t count);

* fd: File descriptor.
* buf: Data to write.
* count: Number of bytes to write.

**Example:**

c

CopyEdit

#include <fcntl.h>

#include <unistd.h>

#include <stdio.h>

int main() {

int fd = open("example.txt", O\_WRONLY | O\_CREAT, 0644);

if (fd < 0) {

perror("open");

return 1;

}

write(fd, "Hello, Linux File System!\n", 26);

close(fd);

return 0;

}

**4. close()**

**Purpose:**

Closes an opened file descriptor, releasing system resources.

**➤ Syntax:**

c

CopyEdit

int close(int fd);

**➤ Example:**

c

CopyEdit

#include <fcntl.h>

#include <unistd.h>

int main() {

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

if (fd >= 0) {

close(fd);

}

return 0;

}

The system calls open(), read(), write(), and close() form the backbone of **low-level file handling in Linux**. They allow:

* Direct access to files using file descriptors,
* Flexible manipulation of file content,
* Efficient resource management.

Understanding these calls is essential for working with Linux at the system level, particularly in applications requiring high performance or direct file control.

3. Device Management System calls

read(), write(), ioctl(), select().

**1. read()**

**➤ Purpose:**

Reads raw data from a device file or standard input.

**➤ Syntax:**

c

CopyEdit

ssize\_t read(int fd, void \*buf, size\_t count);

* fd: File descriptor of the device.
* buf: Buffer to store the data.
* count: Maximum number of bytes to read.

**➤ Example: Read input from terminal (/dev/tty)**

c

CopyEdit

#include <fcntl.h>

#include <unistd.h>

#include <stdio.h>

int main() {

char buffer[100];

int fd = open("/dev/tty", O\_RDONLY);

read(fd, buffer, sizeof(buffer));

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

close(fd);

return 0;

}

**2. write()**

**➤ Purpose:**

Sends raw data to a device file or terminal.

**➤ Syntax:**

c

CopyEdit

ssize\_t write(int fd, const void \*buf, size\_t count);

**➤ Example: Write directly to the terminal**

c

CopyEdit

#include <fcntl.h>

#include <unistd.h>

int main() {

int fd = open("/dev/tty", O\_WRONLY);

write(fd, "Hello from device write!\n", 25);

close(fd);

return 0;

}

**3. ioctl()**

**➤ Purpose:**

Performs **device-specific control operations** that cannot be expressed using regular system calls. It communicates with device drivers for configuration or querying.

**➤ Syntax:**

c

CopyEdit

int ioctl(int fd, unsigned long request, ...);

**➤ Example: Get terminal window size**

c

CopyEdit

#include <stdio.h>

#include <sys/ioctl.h>

#include <unistd.h>

int main() {

struct winsize w;

ioctl(STDOUT\_FILENO, TIOCGWINSZ, &w);

printf("Rows: %d, Cols: %d\n", w.ws\_row, w.ws\_col);

return 0;

}

**4. select()**

**➤ Purpose:**

Waits for multiple file descriptors (like device inputs) to become ready for I/O operations. Commonly used for monitoring multiple devices or sockets.

**➤ Syntax:**

c

CopyEdit

int select(int nfds, fd\_set \*readfds, fd\_set \*writefds, fd\_set \*exceptfds, struct timeval \*timeout);

**➤ Example: Wait for input from stdin**

c

CopyEdit

#include <stdio.h>

#include <unistd.h>

#include <sys/select.h>

int main() {

fd\_set rfds;

FD\_ZERO(&rfds);

FD\_SET(0, &rfds); // STDIN

printf("Waiting for input (5 seconds timeout)...\n");

struct timeval tv = {5, 0}; // 5 seconds timeout

int retval = select(1, &rfds, NULL, NULL, &tv);

if (retval == -1)

perror("select()");

else if (retval)

printf("Data is available on stdin.\n");

else

printf("No data within 5 seconds.\n");

return 0;

}

The device management system calls allow user-space programs to:

* Interact with low-level device interfaces.
* Perform input/output operations directly on hardware.
* Control and monitor device behavior.

4. Network Management System calls

socket(), connect(), send(), recv().

Linux supports **socket programming** through system calls that enable **inter-process communication (IPC)** over a network. These are primarily used to:

* Create endpoints for communication
* Connect to servers
* Send and receive data over TCP/IP or UDP protocols

Networking system calls are essential in client-server applications.

**1. socket()**

**➤ Purpose:**

Creates a new socket (an endpoint for communication).

**➤ Syntax:**

c

CopyEdit

int socket(int domain, int type, int protocol);

* domain: Communication domain (e.g., AF\_INET for IPv4)
* type: Type of service (SOCK\_STREAM for TCP, SOCK\_DGRAM for UDP)
* protocol: Usually set to 0 to choose the default for the given type

**➤ Example:**

c

CopyEdit

int sockfd = socket(AF\_INET, SOCK\_STREAM, 0);

**2. connect()**

**➤ Purpose:**

Connects the client socket to the server’s address and port.

**➤ Syntax:**

c

CopyEdit

int connect(int sockfd, const struct sockaddr \*addr, socklen\_t addrlen);

**➤ Example (Client-side):**

c

CopyEdit

#include <stdio.h>

#include <string.h>

#include <arpa/inet.h>

#include <unistd.h>

int main() {

int sock = socket(AF\_INET, SOCK\_STREAM, 0);

struct sockaddr\_in server;

server.sin\_family = AF\_INET;

server.sin\_port = htons(8080);

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

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

perror("Connection failed");

return 1;

}

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

close(sock);

return 0;

}

**3. send()**

**➤ Purpose:**

Sends data to the connected socket.

**➤ Syntax:**

c

CopyEdit

ssize\_t send(int sockfd, const void \*buf, size\_t len, int flags);

* sockfd: Socket file descriptor
* buf: Buffer containing the message
* len: Length of the message
* flags: Usually 0 for default

**➤ Example:**

c

CopyEdit

char \*message = "Hello from client!";

send(sock, message, strlen(message), 0);

**4. recv()**

**➤ Purpose:**

Receives data from the connected socket.

**➤ Syntax:**

c

CopyEdit

ssize\_t recv(int sockfd, void \*buf, size\_t len, int flags);

**➤ Example:**

c

CopyEdit

char buffer[1024] = {0};

recv(sock, buffer, sizeof(buffer), 0);

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

c

CopyEdit

#include <stdio.h>

#include <string.h>

#include <unistd.h>

#include <arpa/inet.h>

int main() {

int sock = socket(AF\_INET, SOCK\_STREAM, 0);

struct sockaddr\_in server;

server.sin\_family = AF\_INET;

server.sin\_port = htons(8080);

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

connect(sock, (struct sockaddr\*)&server, sizeof(server));

char \*msg = "Hello, Server!";

send(sock, msg, strlen(msg), 0);

char buffer[1024] = {0};

recv(sock, buffer, sizeof(buffer), 0);

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

close(sock);

return 0;

}

The system calls socket(), connect(), send(), and recv() are the core building blocks for network communication in Linux:

* socket() sets up the communication endpoint.
* connect() establishes the link to a remote host.
* send() transmits data.
* recv() receives incoming data.

5. System Information Management System calls

getpid(), getuid(), gethostname(), sysinfo().

System information system calls provide data about the **process**, **user**, and **system** running the Linux OS. These are crucial for monitoring, access control, and debugging. Unlike I/O or network calls, these do not interact with external files or devices but query the **kernel's internal state**.

**1. getpid()**

**Purpose**

Returns the **Process ID (PID)** of the calling process.

**Syntax**

c

CopyEdit

pid\_t getpid(void);

**Example**

c

CopyEdit

#include <stdio.h>

#include <unistd.h>

int main() {

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

return 0;

}

**2. getuid()**

**Purpose**

Returns the **User ID (UID)** of the user running the process.

**Syntax**

c

CopyEdit

uid\_t getuid(void);

**Example**

c

CopyEdit

#include <stdio.h>

#include <unistd.h>

int main() {

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

return 0;

}

📌 Note: UID = 0 usually refers to the root user.

**3. gethostname()**

**Purpose**

Retrieves the **network hostname** of the system.

**Syntax**

c

CopyEdit

int gethostname(char \*name, size\_t len);

* name: Buffer to hold the hostname.
* len: Length of the buffer.

**Example**

c

CopyEdit

#include <stdio.h>

#include <unistd.h>

int main() {

char hostname[1024];

gethostname(hostname, sizeof(hostname));

printf("Hostname: %s\n", hostname);

return 0;

}

**4. sysinfo()**

**Purpose**

Provides detailed **system statistics** like uptime, RAM usage, load average, etc.

**Syntax**

c

CopyEdit

int sysinfo(struct sysinfo \*info);

Requires <sys/sysinfo.h>.

**Example**

c

CopyEdit

#include <stdio.h>

#include <sys/sysinfo.h>

int main() {

struct sysinfo info;

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

printf("Uptime: %ld seconds\n", info.uptime);

printf("Total RAM: %lu MB\n", info.totalram / (1024 \* 1024));

printf("Free RAM: %lu MB\n", info.freeram / (1024 \* 1024));

printf("Number of processes: %u\n", info.procs);

} else {

perror("sysinfo");

}

return 0;

}

System information system calls help developers and system administrators:

* Track process identity (getpid())
* Authenticate users (getuid())
* Gather system-wide data (sysinfo())
* Identify host in a network (gethostname())

These calls offer a window into the **internal state of the Linux system**, vital for performance monitoring, logging, and administrative tasks.

To Submit: Write up for the exhaustive study of the above mentioned system call categories with

their examples.