1.⚙️ Process Management System Calls

a. `fork()`

Creates a new process by duplicating the current process.

```
#include <unistd.h>
#include <stdio.h>

int main() {
    pid_t pid = fork();
    if (pid == 0)
        printf("Child process\n");
    else
        printf("Parent process\n");
    return 0;
}
```

b. `exec()`

Replaces the current process image with a new process image.

```
#include <unistd.h>

int main() {
    char *args[] = {"/bin/ls", "-l", NULL};
    execvp(args[0], args);
    return 0;
}
```

c. `wait()`

Waits for a child process to terminate.

```
int main() {
    pid_t pid = fork();
    if (pid == 0) {
        printf("Child process\n");
    } else {
        wait(NULL);
        printf("Parent waited for child\n");
    }
    return 0;
}
```

d. `exit()`

Terminates the current process.

```
int main() {
    exit(0); // Exits the process successfully
}
```

2. 📁 File Management System Calls

a. `open()`

Opens a file descriptor.

```
int main() {
    int fd = open("test.txt", O_RDONLY);
    if (fd != -1)
        printf("File opened successfully\n");
    return 0;
}
```

b. `read()`

Reads data from a file descriptor.

```
int main() {
    char buffer[100];
    int fd = open("test.txt", O_RDONLY);
    int n = read(fd, buffer, 100);
    buffer[n] = '\0';
    printf("Content: %s\n", buffer);
    close(fd);
    return 0;
}
```

c. `write()`

Writes data to a file descriptor.

```
int main() {
    int fd = open("test.txt", O_WRONLY | O_CREAT, 0644);
    write(fd, "Hello, World!", 13);
    close(fd);
    return 0;
}
```

d. `close()`

Closes a file descriptor.

```
int main() {
    int fd = open("test.txt", O_RDONLY);
    close(fd);
    return 0;
}
```

3. 🎛️ Device Management System Calls

These often involve direct interaction with devices, generally through file descriptors or ioctl interface.

a. `read()` / `write()` (as above)

Used for reading/writing to device files like `/dev/null`, `/dev/tty`, etc.

b. `ioctl()`

Performs device-specific input/output operations.

```
int main() {
    int fd = open("/dev/tty", O_RDONLY);
    int result;
    ioctl(fd, TIOCGWINSZ, &result);
    printf("IOCTL result: %d\n", result);
    close(fd);
    return 0;
}
```

c. `select()`

Monitors multiple file descriptors to see if they are ready for I/O.

```
int main() {
    fd_set readfds;
    FD_ZERO(&readfds);
    FD_SET(0, &readfds); // monitor stdin

    select(1, &readfds, NULL, NULL, NULL);

    if (FD_ISSET(0, &readfds)) {
        printf("Data available on stdin\n");
    }
    return 0;
}
```

4. 🌐 Network Management System Calls

a. `socket()`

Creates a socket.

```
#include <sys/socket.h>
#include <netinet/in.h>

int main() {
    int sockfd = socket(AF_INET, SOCK_STREAM, 0);
    if (sockfd >= 0)
```

```
        printf("Socket created\n");
    return 0;
}
```

b. `connect()`

Connects the socket to a remote host.

```
#include <arpa/inet.h>
int main() {
    int sockfd = socket(AF_INET, SOCK_STREAM, 0);
    struct sockaddr_in serv_addr;
    serv_addr.sin_family = AF_INET;
    serv_addr.sin_port = htons(8080);
    inet_pton(AF_INET, "127.0.0.1", &serv_addr.sin_addr);

    connect(sockfd, (struct sockaddr*)&serv_addr, sizeof(serv_addr));
    return 0;
}
```

c. `send()` and `recv()`

Send and receive data over sockets.

```
#include <sys/socket.h>

int main() {
    int sockfd = socket(AF_INET, SOCK_STREAM, 0);
    char msg[] = "Hello";
    send(sockfd, msg, sizeof(msg), 0);
    char buffer[1024];

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

    return 0;

}
```

5. 🖥️ System Information Management System Calls

a. `getpid()`

Returns process ID of calling process.

```
#include <unistd.h>

#include <stdio.h>


int main() {

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

    return 0;

}
```

b. `getuid()`

Returns user ID of the calling process.

```
int main() {

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

    return 0;

}
```

c. `gethostname()`

Gets the name of the current host.

```
int main() {
    char hostname[1024];

    gethostname(hostname, 1024);

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

    return 0;

}
```

d. `sysinfo()`

Provides system statistics.

```
#include <sys/sysinfo.h>

#include <stdio.h>



int main() {

    struct sysinfo info;

    sysinfo(&info);

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

    return 0;

}
```

CONCLUSION

This study explored various categories of Linux system calls, each serving a specific purpose:

- **Process Control** with `fork`, `exec`, etc.
- **File/Device I/O** with `read`, `write`, `open`, etc.
- **Networking** with `socket`, `send`, `recv`.
- **System Info** retrieval with `getpid`, `getuid`, etc.

These system calls form the core interface between user applications and the Linux kernel.