Understanding Linux System Calls in C Programming
A Tutorial for Operating Systems Students

Introduction
System calls provide the interface between user applications and the operating system kernel. In Linux, system calls are typically wrapped by the C library (glibc) to provide a more convenient programming interface. This tutorial will explore different categories of system calls based on their calling behavior and parameter passing mechanisms.

1. Basic System Calls with Simple Parameters
These system calls take simple numeric or pointer arguments and return a single value.

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

int main() {
    // Open system call - returns a file descriptor
    int fd = open("test.txt", O_RDONLY);
    if (fd < 0) {
        perror("open failed");
        return 1;
    }
    // Close system call - takes a single integer argument
    if (close(fd) < 0) {
        perror("close failed");
        return 1;
    }
    // Get process ID - no arguments
    pid_t pid = getpid();
    printf("Process ID: %d\n", pid);
    return 0;
}
```

2. System Calls with Buffer Parameters
These system calls involve data transfer between user space and kernel space using buffers.

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

int main() {
    char buffer[1024];
    int fd = open("input.txt", O_RDONLY);
    if (fd < 0) {
        perror("open failed");
        return 1;
    }
    // Read system call - fills a user-space buffer
    ssize_t bytes_read = read(fd, buffer, sizeof(buffer));
    if (bytes_read < 0) {
        perror("read failed");
        close(fd);
        return 1;
    }
    // Write system call - transfers data from a buffer
    ssize_t bytes_written = write(STDOUT_FILENO, buffer, bytes_read);
    if (bytes_written < 0) {
        perror("write failed");
        close(fd);
        return 1;
    }
    close(fd);
    return 0;
}
```

3. System Calls with Structure Parameters
These system calls use structures to pass multiple related parameters.

```
#include <sys/stat.h>
#include <time.h>
#include <stdio.h>

int main() {
    struct stat file_info;

    // stat system call fills a structure with file information
    if (stat("test.txt", &file_info) < 0) {
        perror("stat failed");
        return 1;
    }
    printf("File size: %ld bytes\n", file_info.st_size);
    printf("Last modified: %s", ctime(&file_info.st_mtime));
    // Setting file times using structure
    struct timespec times[2];
    times[0].tv_sec = time(NULL);  // Access time
    times[0].tv_nsec = 0;
    times[1].tv_sec = time(NULL);  // Modify time
    times[1].tv_nsec = 0;
    if (utimensat(AT_FDCWD, "test.txt", times, 0) < 0) {
        perror("utimensat failed");
        return 1;
    }
    return 0;
}
```

4. System Calls with Variable Arguments
Some system calls can take a variable number of arguments or complex parameter combinations.

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

int main() {
```

```
    // fcntl can take different numbers and types
of arguments
    // depending on the command
    int fd = open("test.txt", O_RDWR);
    if (fd < 0) {
        perror("open failed");
        return 1;
    }
    // Get file flags. Check flag < 0 for error
    int flags = fcntl(fd, F_GETFL);
    if (flags < 0) {
        perror("fcntl F_GETFL failed");
        close(fd);
        return 1;
    }
    // Set file flags (adding O_APPEND)
    if (fcntl(fd, F_SETFL, flags | O_APPEND) < 0)
{
        perror("fcntl F_SETFL failed");
        close(fd);
        return 1;
    }
    close(fd);
    return 0;
}
```

5. System Calls with Memory Management
These system calls handle memory allocation and mapping operations.

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

int main() {
    // Get page size
    long page_size = sysconf(_SC_PAGE_SIZE);
    // mmap system call - maps memory pages
    void *addr = mmap(NULL, page_size,
                      PROT_READ | PROT_WRITE,
                      MAP_PRIVATE | MAP_ANONYMOUS,
                      -1, 0);
    if (addr == MAP_FAILED) {
        perror("mmap failed");
        return 1;
    }
    // Use the mapped memory
    char *buffer = (char *)addr;
    sprintf(buffer, "Hello, mapped memory!");
    printf("Buffer contains: %s\n", buffer);

    // munmap system call - unmap memory
    if (munmap(addr, page_size) < 0) {
        perror("munmap failed");
        return 1;
    }
    return 0;
}
```

Common Errors and Best Practices

1. **Always Check Return Values**
   - System calls can fail for many reasons
   - Never assume success
   - Use perror() or strerror() to print meaningful error messages

2. **Resource Management**
   - Always close file descriptors
   - Free allocated memory
   - Use cleanup code in error paths

3. **Buffer Handling**
   - Be careful with buffer sizes
   - Check for buffer overflows
   - Handle partial reads/writes

4. **Error Handling Pattern**

```
if (system_call() < 0) {
    perror("system_call failed");
    // Cleanup resources
    return 1;
}
```

Compilation and Running
To compile programs using system calls:

```
$gcc -Wall -o program program.c
```

The `-Wall` flag enables important warning messages that can help catch common mistakes.

Key Points to Remember

1. System calls are the fundamental interface to the kernel
2. Different system calls have different parameter passing mechanisms
3. Error checking is crucial for robust programs
4. Understanding buffer management is essential
5. Memory management system calls require careful handling
6. Resource cleanup is important for preventing leaks