Assignment 4

Comprehensive study of different categories of Linux system calls

Linux system calls serve as the interface between user applications and the kernel. They allow programs to request services from the operating system kernel.

1. Process Management System Calls

Process management system calls handle the creation, execution, and termination of processes.

fork()

The fork() system call creates a new process by duplicating the calling process. This creates a child process that is an exact copy of the parent process except for the return value of the fork() call.

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

int main() {
    pid_t pid = fork();
    if (pid < 0) {
        // Error occurred
        fprintf(stderr, "Fork failed\n");
        return 1;
    } else if (pid == 0) {
        // Child process
        printf("Child process: PID = %d\n", getpid());
    } else {
        // Parent process
        printf("Parent process: PID = %d, Child PID = %d\n", getpid(), pid);
```

```
    }

    return 0;
}
```

exec()

The exec() family of system calls replaces the current process image with a new process image. This replaces the current process with the ls -l command.

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

int main() {
    printf("Before exec()\n");

    char *args[] = {"ls", "-l", NULL};
    execvp("ls", args);

    // This line will only execute if execvp fails
    printf("This won't be printed if execvp succeeds\n");

    return 0;
}
```

wait()

The wait() system call suspends execution of the calling process until one of its children terminates. This shows how a parent process can wait for a child to terminate and retrieve its exit status.

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

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

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

    if (pid == 0) {
        // Child process
        printf("Child process running\n");
        sleep(2);
        printf("Child process exiting\n");
        return 42; // Exit code
    } else {
        // Parent process
        int status;
        printf("Parent waiting for child to terminate\n");
        wait(&status);

        if (WIFEXITED(status)) {
            printf("Child exited with status: %d\n", WEXITSTATUS(status));
        }
    }

    return 0;
}
```

exit()

The exit() system call terminates the calling process.

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

int main() {
   printf("Process starting...\n");

   // Do some work
   printf("Work completed\n");

   // Exit with status code 0 (success)
   exit(0);

   // This line will never be executed
   printf("This won't be printed\n");

   return 1;
}
```

2. File Management System Calls

File management system calls handle file operations like opening, reading, writing, and closing files.

open()

The open() system call opens a file specified by the pathname. This opens (or creates) a file named "example.txt" for writing.

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

```
#include <unistd.h>
int main() {
   int fd = open("example.txt", O_CREAT | O_WRONLY | O_TRUNC, 0644);

   if (fd == -1) {
      perror("Error opening file");
      return 1;
   }
   printf("File opened successfully with descriptor: %d\n", fd);
   close(fd);
   return 0;
}
```

read()

The read() system call reads data from a file descriptor. This reads up to 99 bytes from "example.txt" and displays them.

```
#include <fcntl.h>
#include <stdio.h>
#include <unistd.h>
int main() {
   int fd = open("example.txt", O_RDONLY);

   if (fd == -1) {
      perror("Error opening file");
      return 1;
   }

   char buffer[100];
```

```
    ssize_t bytes_read = read(fd, buffer, sizeof(buffer) - 1);
    if (bytes_read == -1) {
        perror("Error reading file");
        close(fd);
        return 1;
    }

    // Null-terminate the buffer
    buffer[bytes_read] = '\0';
    printf("Read %ld bytes: %s\n", bytes_read, buffer);
    close(fd);
    return 0;
}
```

write()

The write() system call writes data to a file descriptor. This writes a string to "example.txt".

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

int main() {
    int fd = open("example.txt", O_CREAT | O_WRONLY | O_TRUNC, 0644);

    if (fd == -1) {
        perror("Error opening file");
        return 1;
    }
```

```
    const char *message = "Hello, Linux System Calls!";
    ssize_t bytes_written = write(fd, message, strlen(message));

    if (bytes_written == -1) {
        perror("Error writing to file");
        close(fd);
        return 1;
    }
    printf("Wrote %ld bytes to file\n", bytes_written);
    close(fd);
    return 0;
}
```

close()

The close() system call closes a file descriptor.

```
#include <fcntl.h>
#include <stdio.h>
#include <unistd.h>
int main() {
    int fd = open("example.txt", O_RDONLY);
    if (fd == -1) {
        perror("Error opening file");
        return 1;
    }
    printf("File opened successfully with descriptor: %d\n", fd);
    if (close(fd) == -1) {
```

```
        perror("Error closing file");
        return 1;
    }
    printf("File closed successfully\n");
    return 0;
}
```

3. Device Management System Calls

Device management system calls interact with devices connected to the system. Many of these overlap with file operations since Linux treats devices as files.

read() and write() for devices

In Linux, devices are treated as files, so read() and write() are also used for device I/O:

```
#include <fcntl.h>
#include <stdio.h>
#include <unistd.h>
int main() {
    // Open a device file (e.g., a serial port)
    int fd = open("/dev/ttyS0", O_RDWR);
    if (fd == -1) {
        perror("Error opening device");
        return 1;
    }
    // Write to the device
    const char *message = "AT\r\n";
    write(fd, message, 4);
```

```
    // Read from the device
    char buffer[100];
    ssize_t bytes_read = read(fd, buffer, sizeof(buffer) - 1);
    if (bytes_read > 0) {
        buffer[bytes_read] = '\0';
        printf("Response: %s\n", buffer);
    }
    close(fd);
    return 0;
}
```

ioctl()

The ioctl() system call performs device-specific operations that cannot be expressed by regular system calls. This uses ioctl() to eject a CD-ROM tray.

```
#include <fcntl.h>
#include <stdio.h>
#include <sys/ioctl.h>
#include <linux/cdrom.h>
#include <unistd.h>

int main() {
    int fd = open("/dev/cdrom", O_RDONLY | O_NONBLOCK);
    if (fd == -1) {
        perror("Error opening CD-ROM device");
        return 1;
    }
    // Eject the CD-ROM
    if (ioctl(fd, CDROMEJECT) == -1) {
```

```
        perror("Error ejecting CD-ROM");
        close(fd);
        return 1;
    }
    printf("CD-ROM ejected successfully\n");

    close(fd);
    return 0;
}
```

select()

The select() system call monitors multiple file descriptors, waiting for one or more to become ready for I/O operations. This waits for up to 5 seconds for user input, demonstrating non-blocking I/O.

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

int main() {
    fd_set readfds;
    struct timeval tv;
    int ret;
    FD_ZERO(&readfds);
    FD_SET(STDIN_FILENO, &readfds);
    tv.tv_sec = 5;
```

```
    tv.tv_usec = 0;

    printf("Waiting for input for up to 5 seconds...\n");
    ret = select(STDIN_FILENO + 1, &readfds, NULL, NULL, &tv);

    if (ret == -1) {
        perror("select");
        return 1;
    } else if (ret == 0) {
        printf("Timeout occurred, no input available\n");
    } else {
        if (FD_ISSET(STDIN_FILENO, &readfds)) {
            char buffer[100];
            fgets(buffer, sizeof(buffer), stdin);
            printf("Input received: %s", buffer);
        }
    }
    return 0;
}
```

4. Network Management System Calls

Network management system calls handle network communications.

socket()

The socket() system call creates an endpoint for communication. This creates a TCP/IP socket.

```
#include <stdio.h>
#include <sys/socket.h>
#include <arpa/inet.h>
```

```
#include <unistd.h>


int main() {
   int socket_desc = socket(AF_INET, SOCK_STREAM, 0);
   if (socket_desc == -1) {
      perror("Could not create socket");
      return 1;
   }
   printf("Socket created successfully\n");
   close(socket_desc);


   return 0;
}
```

connect()

The connect() system call connects a socket to a specified address. This connects to a remote server on port 80.

```
#include <stdio.h>
#include <sys/socket.h>
#include <arpa/inet.h>
#include <unistd.h>
#include <string.h>


int main() {
   int socket_desc;
   struct sockaddr_in server;


   // Create socket
```

```
    socket_desc = socket(AF_INET, SOCK_STREAM, 0);
    if (socket_desc == -1) {
        perror("Could not create socket");
        return 1;
    }

    server.sin_addr.s_addr = inet_addr("142.250.69.142"); // Example IP
    server.sin_family = AF_INET;
    server.sin_port = htons(80); // HTTP port

    // Connect to remote server
    if (connect(socket_desc, (struct sockaddr *)&server, sizeof(server)) < 0) {
        perror("Connect failed");
        return 1;
    }

    printf("Connected successfully\n");

    close(socket_desc);

    return 0;
}
```

send()

The send() system call sends data through a connected socket. This sends an HTTP GET request to example.com.

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

```
#include <arpa/inet.h>
#include <unistd.h>
#include <string.h>

int main() {
    int socket_desc;
    struct sockaddr_in server;
    char *message = "GET / HTTP/1.1\r\nHost: example.com\r\n\r\n";

    // Create socket
    socket_desc = socket(AF_INET, SOCK_STREAM, 0);
    if (socket_desc == -1) {
        perror("Could not create socket");
        return 1;
    }

    server.sin_addr.s_addr = inet_addr("93.184.216.34"); // example.com IP
    server.sin_family = AF_INET;
    server.sin_port = htons(80);

    // Connect to remote server
    if (connect(socket_desc, (struct sockaddr *)&server, sizeof(server)) < 0) {
        perror("Connect failed");
        return 1;
    }

    // Send HTTP request
    if (send(socket_desc, message, strlen(message), 0) < 0) {
```

```
    perror("Send failed");
    return 1;
  }
  printf("Data sent successfully\n");
  close(socket_desc);
  return 0;
}
```

recv()

The recv() system call receives data from a connected socket. This sends an HTTP request and receives the response.

```
#include <stdio.h>
#include <sys/socket.h>
#include <arpa/inet.h>
#include <unistd.h>
#include <string.h>
int main() {
  int socket_desc;
  struct sockaddr_in server;
  char *message = "GET / HTTP/1.1\r\nHost: example.com\r\n\r\n";
  char server_reply[2000];
  // Create socket
  socket_desc = socket(AF_INET, SOCK_STREAM, 0);
  if (socket_desc == -1) {
    perror("Could not create socket");
    return 1;
  }
  server.sin_addr.s_addr = inet_addr("93.184.216.34"); // example.com IP
```

```
    server.sin_family = AF_INET;
    server.sin_port = htons(80);
    // Connect to remote server
    if (connect(socket_desc, (struct sockaddr *)&server, sizeof(server)) < 0) {
        perror("Connect failed");
        return 1;
    }
    // Send HTTP request
    if (send(socket_desc, message, strlen(message), 0) < 0) {
        perror("Send failed");
        return 1;
    }
    // Receive a reply from the server
    if (recv(socket_desc, server_reply, 2000, 0) < 0) {
        perror("recv failed");
        return 1;
    }
    printf("Server reply:\n%s\n", server_reply);
    close(socket_desc);
    return 0;
}
```

5. System Information Management System Calls

System information management system calls retrieve information about the system.

getpid()

The getpid() system call returns the process ID of the calling process.

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

```
#include <unistd.h>


int main() {
   pid_t pid = getpid();
   printf("Current process ID: %d\n", pid);
   return 0;
}
```

getuid()

The getuid() system call returns the real user ID of the calling process. This shows both real and effective user IDs.

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


int main() {
   uid_t uid = getuid();
   printf("Real user ID: %d\n", uid);


   uid_t euid = geteuid();
   printf("Effective user ID: %d\n", euid);


   return 0;
}
```

gethostname()

The gethostname() system call returns the hostname of the current machine.

```
#include <stdio.h>
#include <unistd.h>
int main() {
```

```
    char hostname[1024];
    if (gethostname(hostname, sizeof(hostname)) == -1) {
        perror("gethostname");
        return 1;
    }
    printf("Hostname: %s\n", hostname);
    return 0;
}
```

sysinfo()

The sysinfo() system call returns system information. This retrieves various system information including uptime, RAM, and process count.

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

int main() {
    struct sysinfo info;

    if (sysinfo(&info) == -1) {
        perror("sysinfo");
        return 1;
    }
    printf("Uptime: %ld seconds\n", info.uptime);
    printf("Total RAM: %lu bytes\n", info.totalram);
    printf("Free RAM: %lu bytes\n", info.freeram);
    printf("Number of processes: %d\n", info.procs);
```

```
    return 0;
}
```

Summary

Linux system calls provide the essential interface between user applications and the kernel, allowing programs to perform a wide range of operations:

1. **Process Management**: Creating, controlling, and terminating processes with calls like fork(), exec(), wait(), and exit().
2. **File Management**: Handling file operations with open(), read(), write(), and close().
3. **Device Management**: Interacting with hardware devices using read(), write(), ioctl(), and select().
4. **Network Management**: Enabling network communications with socket(), connect(), send(), and recv().
5. **System Information Management**: Retrieving system information using getpid(), getuid(), gethostname(), and sysinfo().

Understanding these system calls is crucial for developing efficient and effective Linux applications, as they provide the fundamental operations that applications use to interact with the operating system and the underlying hardware.