ASSIGNMENT 4

To study and learn about various system calls

Introduction to Linux System Calls

System calls are a fundamental mechanism in Linux that enables user-space applications to interact with the kernel. They provide a standardized interface for accessing kernel functionality, allowing applications to perform privileged operations such as reading/writing files, creating new processes, or interacting with hardware devices. When a user-space program needs to perform a privileged operation, it makes a system call to the kernel, which executes the requested action and returns the result to the application

1. The execution of a system call typically involves the following steps:
2. The application prepares arguments for the system call
3. The library function organizes these arguments in the system call table
4. A trap instruction transitions from user space to kernel space
5. The kernel performs the requested operation
6. The result is returned to the user-space application

System calls ensure compatibility and interoperability between applications and the operating system by providing a defined interface for accessing kernel functionality

Process Management System Calls

Process management system calls handle the creation, execution, and termination of processes in the Linux operating system. These calls are essential for multitasking environments where multiple processes need to be managed efficiently.

fork()

The fork() system call creates a new process by duplicating the calling process. This duplication results in two nearly identical processes: the parent (original) process and the child (new) process. The key distinctions between them are their process IDs and the return values they receive from the fork() call
When fork() is executed:
The child process receives a return value of 0
The parent process receives the child's process ID as the return value
The child process runs the same program as the parent process with the same memory image, open files, and environment variables
This system call enables concurrent or parallel execution within programs and is fundamental to the Unix/Linux process model

exec()

The exec() family of system calls (including execve(), execl(), execlp(), etc.) replaces the current process image with a new process image. Unlike fork(), which creates a clone of the existing process, exec() loads an entirely new program into the current process's memory space

When exec() is called:

1. The current process's memory space is cleared
2. A new program is loaded into memory
3. Execution begins at the entry point of the new program
4. The process ID remains unchanged
5. The exec() calls are typically used after a fork() to run a different program in the child process, or to run external programs or scripts from within an application

wait()

The wait() system call allows a parent process to wait for one of its child processes to terminate. When a parent process executes wait(), it is suspended (blocked) until one of its child processes completes execution

Key aspects of wait() include:

It suspends the parent process until a child terminates
It returns the process ID of the terminated child
It also returns the exit status of the child process
It allows for proper synchronization between parent and child processes

This system call is crucial for preventing zombie processes and for implementing sequential execution patterns where a parent needs to wait for a child's completion before proceeding

exit()

The exit() system call terminates the currently executing process and returns a status code to the parent process. This status code can indicate whether the process completed successfully or encountered an error

When exit() is called:

1. The process terminates immediately
2. System resources allocated to the process are released
3. A termination status is returned to the parent process (if it's waiting)
4. The process becomes a zombie until the parent collects its exit status
5. Every process must eventually terminate either through an explicit call to exit() or by returning from the main function

File Management System Calls

File management system calls handle operations related to files, such as opening, reading, writing, and closing files. These calls form the basis of file I/O in Linux systems.

open()

The open() system call is used to open a file and obtain a file descriptor, which is a unique identifier for the opened file. This descriptor is then used in subsequent operations on the file
The syntax for open() is:
int open(const char *path, int flags, ... /* mode_t mode */);
Key parameters include:
path: The pathname of the file to be opened
flags: Specify the access mode (O_RDONLY, O_WRONLY, O_RDWR) and other options
mode: Used only when creating a new file, specifies the permissions
The function returns the smallest available file descriptor, which is a non-negative integer. In case of an error, it returns
Common flags for the open() call include:
1. O_RDONLY: Open for reading only
2. O_WRONLY: Open for writing only
3. O_RDWR: Open for both reading and writing
4.. O_CREAT: Create the file if it doesn't exist
5. O_APPEND: Position the file pointer at the end of the file before each write
The file descriptor returned by open() is subsequently used with read(), write(), and close() system calls

read()

The read() system call is used to read data from an open file into a memory buffer. It takes the file descriptor, buffer address, and the number of bytes to read as parameters
The typical signature of read() is:
ssize_t read(int fd, void *buf, size_t count);
Where:
fd: The file descriptor obtained from open()
buf: A pointer to the buffer where data will be stored
count: The maximum number of bytes to read
The function returns the number of bytes actually read, which may be less than the requested count. A return value of 0 indicates end-of-file, and -1 indicates an error

write()

The write() system call writes data from a memory buffer to an open file. It takes the file descriptor, buffer address, and the number of bytes to write as parameters
The typical signature is:

```
ssize_t write(int fd, const void *buf, size_t count);
```

Where:
fd: The file descriptor obtained from open()
buf: A pointer to the buffer containing data to be written
count: The number of bytes to write
The function returns the number of bytes actually written, which may be less than the requested count. A return value of -1 indicates an error

close()

The close() system call is used to close an open file descriptor, releasing the resources associated with it. Once a file descriptor is closed, it becomes available for reuse in subsequent open() calls
The typical signature is:

```
int close(int fd);
```

Where fd is the file descriptor to be closed. The function returns 0 on success and -1 on error
Closing files is important because the operating system limits the number of files a process can have open simultaneously. Properly closing files prevents resource leaks and ensures that all data is properly written to disk

Device Management System Calls

Device management system calls are used to interact with hardware devices in Linux. These calls treat devices as files (following the "everything is a file" philosophy of Unix/Linux), but with special considerations for hardware interaction.

read() and write() for Devices

In Linux, device files represent physical devices and are located in the /dev directory. The same read() and write() system calls used for regular files are also used for device files, but their behavior depends on the specific device driver implementation
For character devices:
read() retrieves data from the device into a buffer
write() sends data from a buffer to the device
For example, when writing to a device file connected to a modem:

```
int fd = open("/dev/ttyS0", O_WRONLY);write(fd, "AT\r", 3); // Send AT command to the modem
```

This approach allows applications to interact with hardware using the same familiar file I/O interface, maintaining consistency across the operating system

ioctl()

The ioctl() (input/output control) system call provides a way to send device-specific commands to device drivers. It is used when simple read/write operations are not sufficient for controlling a device's behavior
The typical signature is:

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

Where:
fd: The file descriptor for the device
request: A device-specific request code
Additional arguments depending on the request
For example, terminal settings are often controlled with ioctl() calls:

```
struct termios term;ioctl(fd, TCGETS, &term); // Get current terminal settingsterm.c_lflag &= ~ECHO;    // Disable echoioctl(fd, TCSETS, &term); // Apply new settings
```

The ioctl() call is essential for operations that don't fit into the simple read/write model, such as changing device modes, querying device status, or sending specialized commands

select()

The select() system call allows a program to monitor multiple file descriptors, waiting until one or more become ready for some type of I/O operation. This is particularly useful for implementing non-blocking I/O and managing multiple input/output streams concurrently.
The typical signature is:

```
int select(int nfds, fd_set *readfds, fd_set *writefds, fd_set *exceptfds, struct timeval *timeout);
```

Where:
nfds: The highest-numbered file descriptor plus 1
readfds: Set of descriptors to check for reading
writefds: Set of descriptors to check for writing
exceptfds: Set of descriptors to check for exceptions
timeout: Maximum time to wait
The select() call is commonly used in network servers, GUI applications, and any scenario where a program needs to handle multiple input sources without blocking on any single one.

Network Management System Calls

Network management system calls provide the foundation for network programming in Linux, enabling processes to communicate over networks using various protocols.

socket()

The socket() system call creates a communication endpoint (socket) for network communication. It is the first step in establishing network connections.
The typical signature is:

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

Where:
domain: Specifies the protocol family (AF_INET for IPv4, AF_INET6 for IPv6)
type: specifies the communication semantics (SOCK_STREAM for TCP, SOCK_DGRAM for UDP)
protocol: Usually set to 0 for the default protocol
For example, creating a TCP/IP socket:

```
int sock_fd = socket(AF_INET, SOCK_STREAM, 0);
```

The function returns a file descriptor for the new socket, which is then used in subsequent network operations.

connect()

The connect() system call establishes a connection between a socket and a specified address. It is used by client applications to connect to servers.
The typical signature is:

```
int connect(int sockfd, const struct sockaddr *addr, socklen_t addrlen);
```

Where:
sockfd: The socket file descriptor
addr: Points to a sockaddr structure containing the destination address
addrlen: Size of the address structure
For example, connecting to a web server:

```
struct sockaddr_in server_addr;server_addr.sin_family =
AF_INET;server_addr.sin_port = htons(80);inet_pton(AF_INET, "192.168.1.1",
&server_addr.sin_addr);connect(sock_fd, (struct sockaddr *)&server_addr,
sizeof(server_addr));
```

send() and recv()

The send() and recv() system calls are used to transmit and receive data over connected sockets.
The send() function has the signature:

```
ssize_t send(int sockfd, const void *buf, size_t len, int flags);
```

Where:
sockfd: The socket file descriptor
buf: Points to the data to be sent
len: Length of the data in bytes
flags: Special behavior flags (usually 0)
The recv() function has the signature:

```
ssize_t recv(int sockfd, void *buf, size_t len, int flags);
```

Where:
sockfd: The socket file descriptor
buf: Buffer to store received data
len: Maximum length of the buffer
flags: Special behavior flags (usually 0)
These functions return the number of bytes sent or received, or -1 on error.

System Information Management System Calls

System information management system calls provide processes with information about the system and their execution environment.

getpid()

The getpid() system call returns the process ID of the calling process. This unique identifier is assigned by the kernel when the process is created.

The typical signature is:

```
pid_t getpid(void);
```

Example usage:

```
pid_t my_pid = getpid();printf("My process ID is: %d\n", my_pid);
```

getuid()

The getuid() system call returns the real user ID of the calling process. This ID identifies the user who owns the process.
The typical signature is:

```
uid_t getuid(void);
```

Example usage:

```
uid_t my_uid = getuid();printf("My user ID is: %d\n", my_uid);
```

There is also a related call, geteuid(), which returns the effective user ID, potentially different from the real user ID when a program is running with setuid permissions.

gethostname()

The gethostname() system call retrieves the hostname of the current machine.
The typical signature is:

```
int gethostname(char *name, size_t len);
```

Where:
name: Buffer to store the hostname
len: Size of the buffer
Example usage:

```
char hostname[256];gethostname(hostname,
sizeof(hostname));printf("Hostname: %s\n", hostname);
```

sysinfo()

The sysinfo() system call returns information about system statistics and resources, such as memory usage and load averages.
The typical signature is:

```
int sysinfo(struct sysinfo *info);
```

Where info points to a sysinfo structure that will be filled with system information.
Example usage

```
struct sysinfo si;sysinfo(&si);printf("Total RAM: %lu bytes\n",
si.totalram);printf("Free RAM: %lu bytes\n", si.freeram);printf("System uptime: %ld
seconds\n", si.uptime);
```