OPERATING SYSTEMS
LAB ASSIGNMENT 4

Comprehensive study of different categories of Linux system calls, categorized as

1. Process Management System Calls

These system calls allow a process to create, execute, wait for, and terminate child processes. They are fundamental to multitasking in Linux systems.

fork()

The fork() system call is used to create a new process by duplicating the calling process. The new process is referred to as the child process. The child process and the parent process run concurrently. The only difference is the return value:

- Returns 0 to the child process.
- Returns the PID of the child to the parent process.
- Returns -1 if the creation of the child process fails.

Syntax:
pid_t fork(void);

Use Case:
Often used to create a separate process for execution. After fork(), both parent and child can execute different parts of the code.

Example:

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

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

exec()

The exec() family of functions replaces the current process image with a new process image. That is, it loads a new program into the current process space and starts its execution. exec() is usually called after fork() in the child process to run a different program.

Common Variants:

- execl(), execlp(), execle()

- execv(), execvp(), execvpe()

Syntax (example):
int execvp(const char *file, char *const argv[]);

Use Case:
To run a completely different program in a newly created child process.

Example:

```
#include <unistd.h>

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

Note: If exec() is successful, the code after the exec() call will not be executed.

wait()

The wait() system call makes the parent process **pause execution until** one of its child processes terminates. This is crucial to prevent zombie processes and to synchronize the flow of parent and child processes.

Syntax:
pid_t wait(int *status);

Returns:
- Process ID of the terminated child.
- -1 on failure.

Use Case:
Used in the parent process after fork() to ensure the child completes its execution first.

Example:

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

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

```
    else {
        wait(NULL);
        printf("Child finished, parent continues\n");
    }
    return 0;
}
```

exit()

The exit() system call terminates the calling process. It performs all necessary cleanup operations (closing file descriptors, freeing memory, etc.) and returns an **exit status** to the parent process.

Syntax:

```
void exit(int status);
```

Use Case:

To terminate a program cleanly, optionally returning a status code to the OS or to a parent process waiting for it.

Example:

```
#include <stdlib.h>

int main() {
    exit(0);
}
```

2. File Management System calls

These system calls allow programs to interact with files stored on the disk. In Linux, everything is treated as a file, including devices and sockets, so file management calls are essential to a wide range of operations.

open()

The open() system call is used to open a file or device. It returns a file descriptor (an integer), which uniquely identifies the open file within a process. This file descriptor is then used in subsequent calls like read(), write(), and close().

Syntax:

```
int open(const char *pathname, int flags);
```

Common flags:

- O_RDONLY: Open for reading only.
- O_WRONLY: Open for writing only.
- O_RDWR: Open for reading and writing.

- O_CREAT: Create file if it does not exist (used with a third mode argument).

Example:

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

int main() {
   int fd = open("example.txt", O_RDONLY);
   if (fd < 0) {
      perror("Failed to open file");
      return 1;
   }
   printf("File opened with fd: %d\n", fd);
   return 0;
}
```

read()

The read() system call reads data from a file descriptor into a buffer. It attempts to read the specified number of bytes but may return fewer.

Syntax:

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

Returns: Number of bytes read, 0 for EOF, -1 for error.

Example:

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

int main() {
   char buffer[100];
   int fd = open("example.txt", O_RDONLY);
   int bytesRead = read(fd, buffer, sizeof(buffer) - 1);
   buffer[bytesRead] = '\0';
   printf("Read data: %s\n", buffer);
   close(fd);
   return 0;
}
```

write()

The write() system call writes data from a buffer to a file descriptor.

Syntax:

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

Returns: Number of bytes written or -1 on error.

Example:

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

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

close()

The close() system call closes an open file descriptor, freeing the associated resources.

Syntax:

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

Returns: 0 on success, -1 on error.

Example:

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

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

3. Device Management System calls

Device files represent hardware or virtual devices. Linux treats them as special files in /dev/. Device management system calls enable processes to communicate with these hardware interfaces.

read()/write()

These work identically to how they operate on regular files but are used with device file descriptors.

Example :

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

int main() {
   int fd = open("/dev/tty", O_WRONLY);
   write(fd, "Hello Terminal\n", 15);
   close(fd);
   return 0;
}
```

ioctl()

ioctl() stands for input/output control. It is used for device-specific operations not covered by standard system calls. It's often used for setting baud rates, toggling device modes, or querying hardware status.

Syntax:

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

Example:

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

int main() {
   int fd = open("/dev/tty", O_RDONLY);
   int bytes_available;
   ioctl(fd, FIONREAD, &bytes_available);
   printf("Bytes available to read: %d\n", bytes_available);
   close(fd);
   return 0;
}
```

select()

select() is used to monitor multiple file descriptors to see if any are ready for I/O. It is commonly used in device and network programming.

Syntax:

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

Example:

```
#include <sys/select.h>
#include <unistd.h>
#include <stdio.h>
```

```
int main() {
    fd_set rfds;
    struct timeval tv;
    int retval;

    FD_ZERO(&rfds);
    FD_SET(0, &rfds);  // STDIN

    tv.tv_sec = 5;
    tv.tv_usec = 0;

    retval = select(1, &rfds, NULL, NULL, &tv);
    if (retval)
        printf("Data is available now.\n");
    else
        printf("No data within five seconds.\n");

    return 0;
}
```

4. Network Management System calls

Network communication in Linux is handled through sockets. These system calls are vital for creating client-server applications over the Internet.

socket()

socket() creates an endpoint for communication.

Syntax:

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

- AF_INET = IPv4
- SOCK_STREAM = TCP
- SOCK_DGRAM = UDP

Example:

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

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

connect()

Used by a client to connect to a server.

Syntax:

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

Example:

```
#include <arpa/inet.h>
#include <netinet/in.h>

struct sockaddr_in server;
server.sin_family = AF_INET;
server.sin_port = htons(8080);
inet_pton(AF_INET, "127.0.0.1", &server.sin_addr);
connect(sockfd, (struct sockaddr *)&server, sizeof(server));
```

send()

Sends data to a connected socket.

Syntax:

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

Example:

```
send(sockfd, "Hello, server", strlen("Hello, server"), 0);
```

recv()

Receives data from a socket.

Syntax:

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

Example:

```
char buffer[1024];
recv(sockfd, buffer, sizeof(buffer), 0);
```

5. System Information Management System calls

These calls provide system-level information like process IDs, memory usage, and system uptime.

getpid()

Returns the process ID of the calling process.

Example:

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

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

getuid()

Returns the real user ID of the calling process.

Example:

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

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

gethostname()

Retrieves the hostname of the current machine.

Example:

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

char name[1024];
gethostname(name, 1024);
printf("Hostname: %s\n", name);
```

sysinfo()

Provides various system statistics such as uptime, free memory, and load average.

Example:

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

struct sysinfo si;
sysinfo(&si);
printf("System uptime: %ld seconds\n", si.uptime);
printf("Total RAM: %lu MB\n", si.totalram / 1024 / 1024);
```