Course Title:Operating Systems

Course code: BCSE303L Slot: B1+TB1

Digital Assignment

Presented by:

Vaishnavi Gupta (21BCE0189) Leader's Phone Number: 8303513693

Manav Pabari (21BCE3298)

G Pooja Alankrutha Sai (21BCT0338)

Abhinav Garg (21BCE3310)

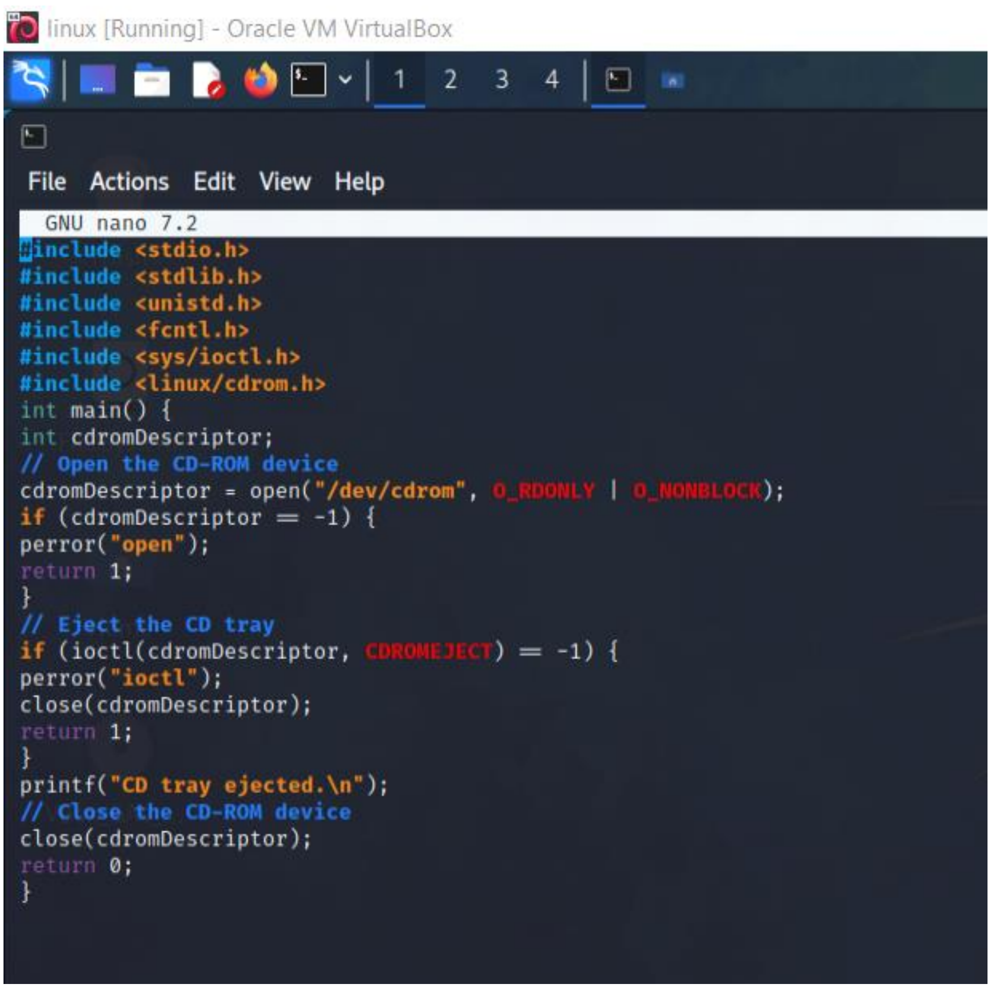
Bhavya Sharma (21BCE2407)

Aim:

Design and Implement minimum five system calls of your own choice. Select appropriate OS for implementation.

We have implemented these 5 system calls using Kali Linux Operating system.

A) Device management system calls:




```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <fcntl.h>
#include <sys/ioctl.h>
#include <linux/cdrom.h>
int main() {
int cdromDescriptor;
// Open the CD-ROM device
cdromDescriptor = open("/dev/cdrom", O_RDONLY | O_NONBLOCK);
if (cdromDescriptor == -1) {
perror("open");
return 1;
}
// Eject the CD tray
if (ioctl(cdromDescriptor, CDROMEJECT) == -1) {
perror("ioctl");
close(cdromDescriptor);
return 1;
}
printf("CD tray ejected.\n");
// Close the CD-ROM device
close(cdromDescriptor);
return 0;
}
```

Here's an advanced program that demonstrates device management system calls, specifically `open()`, `ioctl()`, and `close()`:

This program demonstrates the device management system calls to open a CD-ROM device, eject the CD tray, and close the device.

- Explanation:
  1. The program starts by including necessary header files: `<stdio.h>`, `<stdlib.h>`, `<unistd.h>`, `<fcntl.h>`, `<sys/ioctl.h>`, and `<linux/cdrom.h>`. These headers provide the required functions and constants for device management system calls.
  2. In the `main()` function, we declare a variable `cdromDescriptor` to store the file descriptor of the CD-ROM device.
  3. We use the `open()` system call to open the CD-ROM device `/dev/cdrom` with the following flags:
     - `O_RDONLY`: Open the device for reading
     - `O_NONBLOCK`: Set the file descriptor to non-blocking mode

     The CD-ROM device is typically located at `/dev/cdrom` or `/dev/sr0` in Linux systems.
  4. If the `open()` call returns -1, an error occurred, and we print an error message using `perror()` and return with an error code.
  5. We use the `ioctl()` system call to send an eject command to the CD-ROM device. The `CDROMEJECT` constant from `<linux/cdrom.h>` specifies the eject command.
  6. If the `ioctl()` call returns -1, an error occurred, and we print an error message using `perror()`, close the device using `close()`, and return with an error code.
  7. We print a message indicating that the CD tray has been ejected.
  8. We close the CD-ROM device using the `close()` system call.

- To compile and run the program:

  1. Save the code in a file named `device_management.c`.

  2. Open the terminal and navigate to the directory where the file is saved.

  3. Compile the program using the following command: `gcc device_management.c -device_management`.

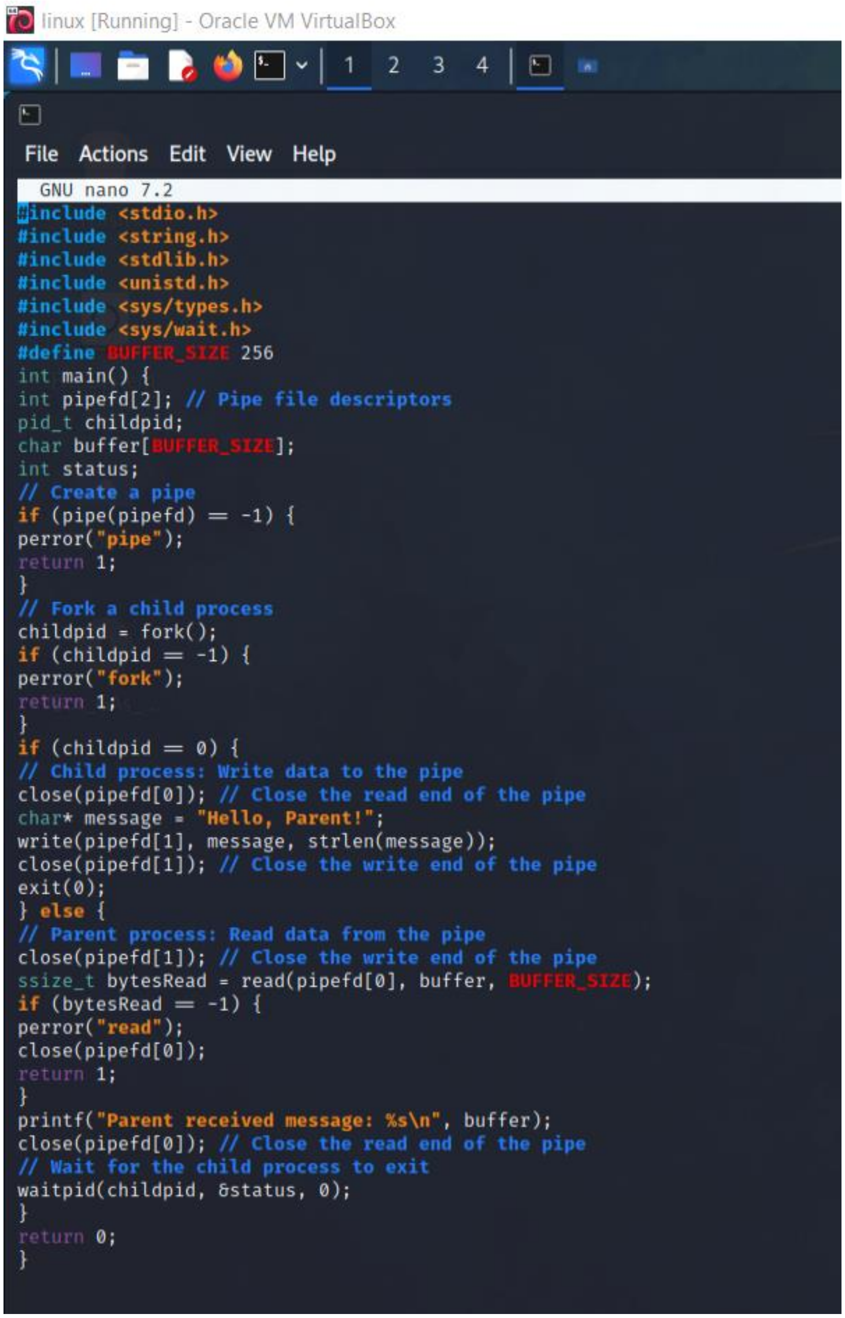
  4. Run the program with root privileges: ` ./device_management`.

- Execution:

B) Communication system calls:

- **<u>Here's an advanced program that demonstrates communication system calls, specifically inter process communication using pipes (`pipe()`) and parent-child process communication:</u>**

- Code:




```c
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/wait.h>
#define BUFFER_SIZE 256
int main() {
int pipefd[2]; // Pipe file descriptors
pid_t childpid;
char buffer[BUFFER_SIZE];
int status;
// Create a pipe
if (pipe(pipefd) == -1) {
perror("pipe");
return 1;
}
// Fork a child process
childpid = fork();
if (childpid == -1) {
perror("fork");
return 1;
}
if (childpid == 0) {
// Child process: Write data to the pipe
close(pipefd[0]); // Close the read end of the pipe
char* message = "Hello, Parent!";
write(pipefd[1], message, strlen(message));
close(pipefd[1]); // Close the write end of the pipe
exit(0);
} else {
// Parent process: Read data from the pipe
close(pipefd[1]); // Close the write end of the pipe
ssize_t bytesRead = read(pipefd[0], buffer, BUFFER_SIZE);
if (bytesRead == -1) {
perror("read");
close(pipefd[0]);
return 1;
}
printf("Parent received message: %s\n", buffer);
close(pipefd[0]); // Close the read end of the pipe
// Wait for the child process to exit
waitpid(childpid, &status, 0);
}
return 0;
}
```

This program demonstrates interprocess communication using pipes. It creates a pipe, forks a child process, and performs communication between the parent and child processes.

- Explanation:

1. The program starts by including necessary header files: `<stdio.h>`, `<stdlib.h>`, `<unistd.h>`, ` <sys/types.h>`, and `<sys/wait.h>`. These headers provide the required functions and data types for interprocess communication.

2. In the `main()` function, we declare variables `pipefd` to store the pipe file descriptors, `childpid` to store the child process ID, `buffer` to read the data from the pipe, and `status` to store the exit status of the child process.

3. We use the `pipe()` system call to create a pipe. The `pipefd` array will hold two file descriptors, ` pipefd[0]` for reading from the pipe and `pipefd[1]` for writing to the pipe.

4. If the `pipe()` call returns -1, an error occurred, and we print an error message using `perror()` and return with an error code.

5. We use the `fork()` system call to create a child process. The `fork()` function returns the child process ID in the parent process and 0 in the child process.

6. If the `fork()` call returns -1, an error occurred, and we print an error message using `perror()` and return with an error code.

7. If the `fork()` call returns 0, we are in the child process. In the child process, we close the read end of the pipe using `close(pipefd[0])`, write a message to the pipe using `write(pipefd[1], message, strlen(message))`, and then close the write end of the pipe using `close(pipefd[1])`.

Finally, we exit the child process using `exit(0)`.

8. If the `fork()` call returns a positive value, we are in the parent process. In the parent process, we close the write end of the pipe using `close(pipefd[1])`, read data from the pipe using `read(pipefd[0], buffer, BUFFER_SIZE)`, and then close the read end of the pipe using `close(pipefd[0])`. We print the received message using `printf()`. Finally, we wait for the child process to exit using `waitpid(childpid, &status, 0)`.

- To Compile and Run the Program:

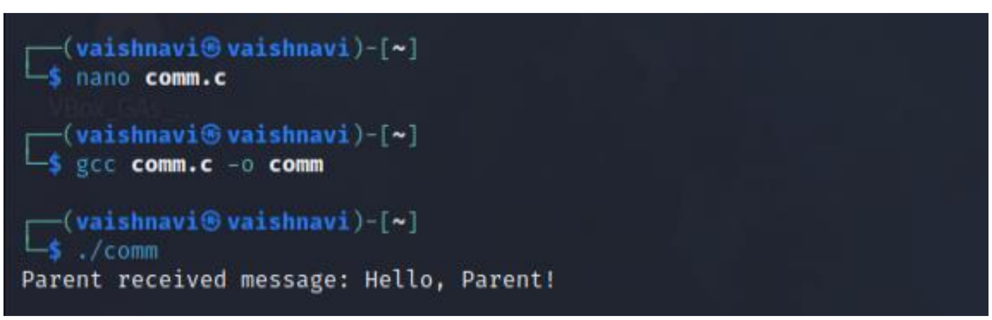
1. Save the code in a file named `communication.c`.

2. Open the terminal and navigate to the directory where the file is saved.

3. Compile the program using the following command: `gcc communication.c -o communication`.

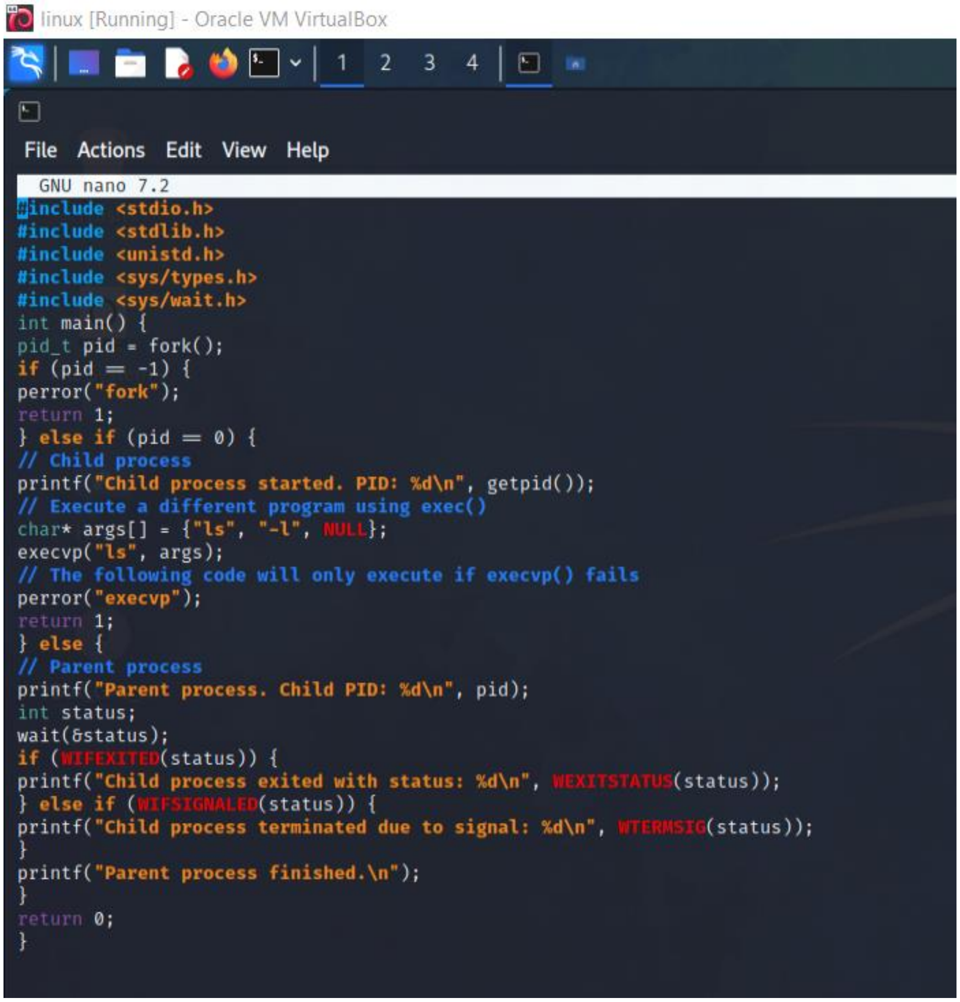
4. Run the program: `./communication`.

- Output:



C) Process System calls:

- **<u>This program demonstrates how to perform interprocess communication using pipes in C, allowing the parent and child processes to exchange data.</u>**
- Code:




```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/wait.h>
int main() {
pid_t pid = fork();
if (pid == -1) {
perror("fork");
return 1;
} else if (pid == 0) {
// Child process
printf("Child process started. PID: %d\n", getpid());
// Execute a different program using exec()
char* args[] = {"ls", "-l", NULL};
execvp("ls", args);
// The following code will only execute if execvp() fails
perror("execvp");
return 1;
} else {
// Parent process
printf("Parent process. Child PID: %d\n", pid);
int status;
wait(&status);
if (WIFEXITED(status)) {
printf("Child process exited with status: %d\n", WEXITSTATUS(status));
} else if (WIFSIGNALED(status)) {
printf("Child process terminated due to signal: %d\n", WTERMSIG(status));
}
printf("Parent process finished.\n");
}
return 0;
}
```

This program demonstrates the use of process control system calls to create a child process, execute a different program within the child process using `exec()`, and handle the exit status of the child process within the parent process.

- Explanation:

1. The program starts by including necessary header files: `<stdio.h>`, `<stdlib.h>`, `<unistd.h>`, `<sys/types.h>`, and `<sys/wait.h>`. These headers provide the required functions and data types for process control system calls.

2. The `main()` function begins, and we declare a variable `pid` of type `pid_t`. The `pid_t` data type represents process IDs.

3. We use the `fork()` system call to create a child process. The return value of `fork()` is stored in the `pid` variable. If `fork()` returns -1, an error occurred, and we print an error message using `perror()` and return with an error code.

4. If `fork()` returns 0, we are inside the child process. We print a message indicating that the child process has started, along with its process ID (`getpid()`).

5. We then use the `execvp()` system call to execute a different program, `ls`, with the argument `-l`. The `execvp()` function replaces the current process image with a new process image specified by the provided program name and arguments. If `execvp()` fails, we print an error message using `perror()` and return with an error code.

6. If `fork()` returns a positive value (the process ID of the child process) in the parent process, we are inside the parent process. We print a message indicating that it's the parent process, along with the child process ID (`pid`).

7. We use the `wait()` system call to wait for the child process to complete. The `wait()` function suspends the execution of the parent process until one of its child processes exits or receives a termination signal.

8. We store the exit status of the child process in the `status` variable.

9. We check the exit status using the macros `WIFEXITED()` and `WEXITSTATUS()`. If the child process terminated normally, `WIFEXITED()` will be true, and we print the exit status using `WEXITSTATUS()`. If the child process terminated due to a signal, `WIFSIGNALED()` will be true, and we print the termination signal using `WTERMSIG()`.

10. Finally, we print a message indicating that the parent process has finished.

11. The `main()` function ends, and the program returns 0, indicating successful execution.

- Execution:

1. The program is compiled and executed.

2. The `fork()` system call is invoked, creating a child process.

3. In the child process, the program prints a message stating that it's the child process and displays its process ID (`getpid()`).

4. The child process uses `execvp()` to execute the `ls -l` command. The output of `ls -l` is displayed on the terminal.

5. If the `execvp()` call fails, an error message is printed.

6. In the parent process, the program prints a message stating that it's the parent process and displays the child process ID (`pid`).

7. The parent process waits for the child process to complete using `wait()`. During this time, the parent process is suspended.

8. Once the child process finishes, the parent process resumes execution.

9. The program determines the exit status of the child process and prints the appropriate message.

➢ Output:

```
┌──(vaishnavi㉿vaishnavi)-[~]
└─$ nano process.c

┌──(vaishnavi㉿vaishnavi)-[~]
└─$ gcc process.c -o process

┌──(vaishnavi㉿vaishnavi)-[~]
└─$ ./process
Parent process. Child PID: 11004
Child process started. PID: 11004
total 188
-rwxr-xr-x 1 vaishnavi vaishnavi 16400 Jun 29 10:47 comm
-rw-r--r-- 1 vaishnavi vaishnavi  1089 Jun 26 01:24 comm.c
drwxr-xr-x 2 vaishnavi vaishnavi  4096 Jun 20 21:12 Desktop
-rwxr-xr-x 1 vaishnavi vaishnavi 16160 Jun 29 10:47 dev_mngmt
-rw-r--r-- 1 vaishnavi vaishnavi   519 Jun 26 01:23 dev_mngmt.c
drwxr-xr-x 2 vaishnavi vaishnavi  4096 Jun 20 21:12 Documents
drwxr-xr-x 2 vaishnavi vaishnavi  4096 Jun 22 02:12 Downloads
-rw------- 1 vaishnavi vaishnavi     7 Jun 23 22:01 example.txt
-rwxr-xr-x 1 vaishnavi vaishnavi 16112 Jun 23 22:01 filecreation
-rw-r--r-- 1 vaishnavi vaishnavi   310 Jun 23 22:01 filecreation.c
-rwxr-xr-x 1 vaishnavi vaishnavi 16256 Jun 29 10:48 file_mngmt
-rw-r--r-- 1 vaishnavi vaishnavi  1030 Jun 26 01:24 file_mngmt.c
-rw-r--r-- 1 vaishnavi vaishnavi    41 Jun 29 10:48 file.txt
-rwxr-xr-x 1 vaishnavi vaishnavi 16320 Jun 29 10:43 info
-rw-r--r-- 1 vaishnavi vaishnavi   848 Jun 26 01:24 info.c
drwxr-xr-x 2 vaishnavi vaishnavi  4096 Jun 20 21:12 Music
drwxr-xr-x 5 vaishnavi vaishnavi  4096 Jun 26 18:27 Pictures
-rwxr-xr-x 1 vaishnavi vaishnavi 16256 Jun 29 10:48 process
-rw-r--r-- 1 vaishnavi vaishnavi   819 Jun 26 01:24 process.c
drwxr-xr-x 2 vaishnavi vaishnavi  4096 Jun 20 21:12 Public
-rwxr-xr-x 1 vaishnavi vaishnavi 16016 Jun 23 21:40 syscall_example
-rw-r--r-- 1 vaishnavi vaishnavi   144 Jun 23 21:40 syscall_example.c
drwxr-xr-x 2 vaishnavi vaishnavi  4096 Jun 20 21:12 Templates
drwxr-xr-x 2 vaishnavi vaishnavi  4096 Jun 20 21:12 Videos
-rwxr-xr-x 1 vaishnavi vaishnavi    10 Jun 20 21:14 v.sh
Child process exited with status: 0
Parent process finished.
```

D) File management system calls:

<u>Here's an advanced program that demonstrates file management system calls, specifically `open()`, `read()`, `write()`, and `close()`:</u>

➢ Code:


```c
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <unistd.h>
#include <fcntl.h>
#define BUFFER_SIZE 256
int main() {
int fileDescriptor;
char buffer[BUFFER_SIZE];
// Open a file for writing
fileDescriptor = open("file.txt", O_WRONLY | O_CREAT | O_TRUNC, 0644);
if (fileDescriptor == -1) {
perror("open");
return 1;
}
// Write data to the file
char* message = "This is OS DA which involves system calls";
ssize_t bytesWritten = write(fileDescriptor, message, strlen(message));
if (bytesWritten == -1) {
perror("write");
close(fileDescriptor);
return 1;
}
// Close the file
close(fileDescriptor);
// Open the file for reading
fileDescriptor = open("file.txt", O_RDONLY);
if (fileDescriptor == -1) {
perror("open");
return 1;
}
// Read data from the file
ssize_t bytesRead = read(fileDescriptor, buffer, BUFFER_SIZE);
if (bytesRead == -1) {
perror("read");
close(fileDescriptor);
return 1;
}
// Close the file
close(fileDescriptor);
// Print the content of the file
printf("File content: %s\n", buffer);
return 0;}
```

- Explanation:

1. The program starts by including necessary header files: `<stdio.h>`, `<stdlib.h>`, `<unistd.h>`, and `<fcntl.h>`. These headers provide the required functions and constants for file management system calls.

2. We define a constant `BUFFER_SIZE` to specify the size of the buffer used for reading and writing data.

3. In the `main()` function, we declare variables `fileDescriptor` for storing the file descriptor and `buffer` for reading the file content.

4. We use the `open()` system call to open a file named "file.txt" with the following flags:

- `O_WRONLY`: Open the file for writing

- `O_CREAT`: Create the file if it doesn't exist

- `O_TRUNC`: Truncate the file to zero length if it already exists

The file is created with file permissions `0644` (readable and writable by the owner, readable by others).

5. If the `open()` call returns -1, an error occurred, and we print an error message using `perror()` and return with an error code.

6. We use the `write()` system call to write the string "Hello, World!" to the file. The `write()` function returns the number of bytes written.

7. If the `write()` call returns -1, an error occurred, and we print an error message using `perror()`, close the file using `close()`, and return with an error code.

8. We close the file using the `close()` system call.

9. We reopen the file using the `open()` system call, this time with the `O_RDONLY` flag to open it for reading.

10. If the `open()` call returns -1, an error occurred, and we print an error message using `perror()` and return with an error code.

11. We use the `read()` system call to read the contents of the file into the `buffer`. The `read()` function returns the number of bytes read.

12. If the `read()` call returns -1, an error occurred, and we print an error message using `perror()`, close the file using `close()`, and return with an error code.

13. We close the file using the `close()` system call.

14. Finally, we print the content of the file by printing the `buffer` contents.

- To Compile and Run the Program:

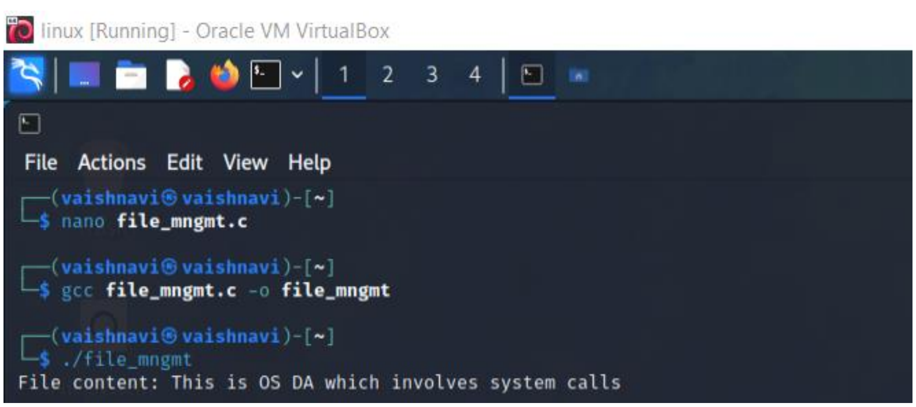
1. Save the code in a file named `file_management.c`.

2. Open the terminal and navigate to the directory where the file is saved.

3. Compile the program using the following command: `gcc file_management.c -o file_management`.

4. Run the program: `./file_management`.

- Output:



E) Information maintenance system calls

- **Here's an advanced program that demonstrates information maintenance system calls, specifically `gethostname()`, `getpid()`, `getuid()`, `getgid()`, `getpwuid()`, and `getgrgid()`:**

- Code:

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
#include <pwd.h>
#include <grp.h>
int main() {
// Get the hostname
char hostname[256];
if (gethostname(hostname, sizeof(hostname)) == -1) {
perror("gethostname");
return 1;
}
printf("Hostname: %s\n", hostname);
// Get the process ID
pid_t pid = getpid();
printf("Process ID: %d\n", pid);
// Get the user ID and username
uid_t uid = getuid();
struct passwd *user_info = getpwuid(uid);
if (user_info == NULL) {
perror("getpwuid");
return 1;
}
printf("User ID: %d\n", uid);
printf("Username: %s\n", user_info->pw_name);
// Get the group ID and group name
gid_t gid = getgid();
struct group *group_info = getgrgid(gid);
if (group_info == NULL) {
perror("getgrgid");
return 1;
}
printf("Group ID: %d\n", gid);
printf("Group Name: %s\n", group_info->gr_name);
return 0;
}
```

- Explanation:

1. The program starts by including necessary header files: `<stdio.h>`, `<stdlib.h>`, `<unistd.h>`, `<sys/types.h>`, `<pwd.h>`, and `<grp.h>`. These headers provide the required functions and data types for information maintenance system calls.

2. In the `main()` function, we declare variables `hostname` to store the hostname, `pid` to store the process ID, `uid` to store the user ID, and `gid` to store the group ID.

3. We use the `gethostname()` system call to retrieve the hostname of the system. The hostname is stored in the `hostname` array.

4. If the `gethostname()` call returns -1, an error occurred, and we print an error message using `perror()` and return with an error code.

5. We print the hostname using `printf()`.

6. We use the `getpid()` system call to retrieve the process ID.

7. We print the process ID using `printf()`.

8. We use the `getuid()` system call to retrieve the user ID of the current process.

9. We use the `getpwuid()` function to retrieve the `passwd` structure for the user associated with the user ID.

10. If the `getpwuid()` call returns `NULL`, an error occurred, and we print an error message using `perror()` and return with an error code.

11. We print the user ID and username using `printf()`.

12. We use the `getgid()` system call to retrieve the group ID of the current process.

13. We use the `getgrgid()` function to retrieve the `group` structure for the group associated with the group ID.

14. If the `getgrgid()` call returns `NULL`, an error occurred, and we print an error message using `perror()` and return with an error code.

15. We print the group ID and group name using `printf()`.

- To Compile and run the program:

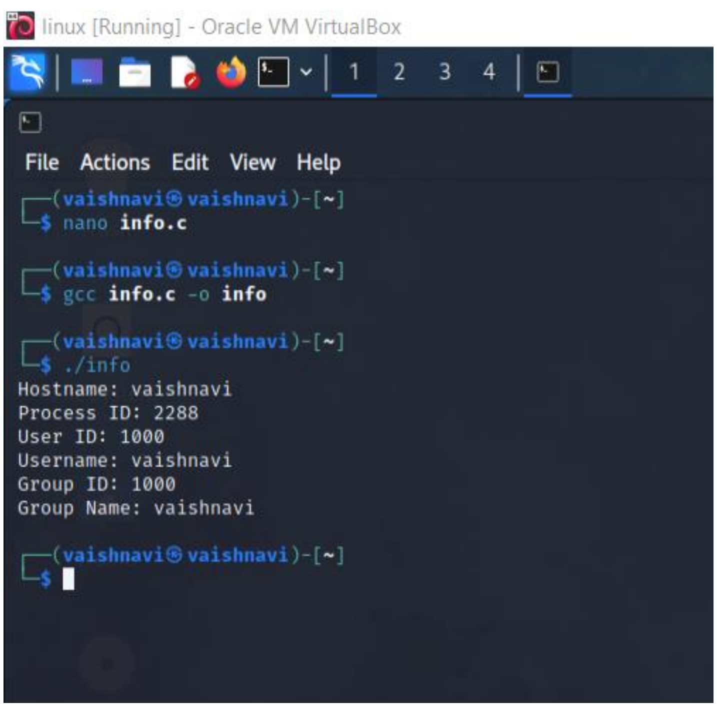
1. Save the code in a file named `information_maintenance.c`.

2. Open the terminal and navigate to the directory where the file is saved.

3. Compile the program using the following command: `gcc information_maintenance.c -o information_maintenance`.

4. Run the program: `./information_maintenance`.

- Output:



Video link: