Batch: C2 **Roll No.: 16010122323**

Experiment No. 03

Grade: AA / AB / BB / BC / CC / CD /DD

TITLE: System calls

AIM: To understand the working Process based system calls.

Expected Outcome of Experiment:

CO 1. To introduce basic concepts and functions of operating systems.

Books/ Journals/ Websites referred:

1. **Silberschatz A., Galvin P., Gagne G. "Operating Systems Principles", Willey Eight edition.**
2. **William Stallings "Operating Systems" Person, Seventh Edition Edition.**
3. **Sumitabha Das " UNIX Concepts & Applications", McGraw Hill Second Edition.**

Pre Lab/ Prior Concepts:

System Calls Provide the Interface between a process and the OS.

System calls are usually made when a process in user mode requires access to a resource.

Then it requests the kernel to provide the resource via a system call.

System calls are required in the following situations −

1. If a file system requires the creation or deletion of files.
2. Reading and writing from files also require a system call.
3. Creation and management of new processes.

4. Network connections also require system calls. This includes sending and receiving packets.
5. Access to a hardware devices such as a printer, scanner etc. requires a system call.

Description of the application to be implemented:

Program for System Call:

1. Write a program using system calls to copy contents of one file into another

2. Implement a program to demonstrate process creation and termination using system calls like fork(), exec(), and wait().

Implementation details:

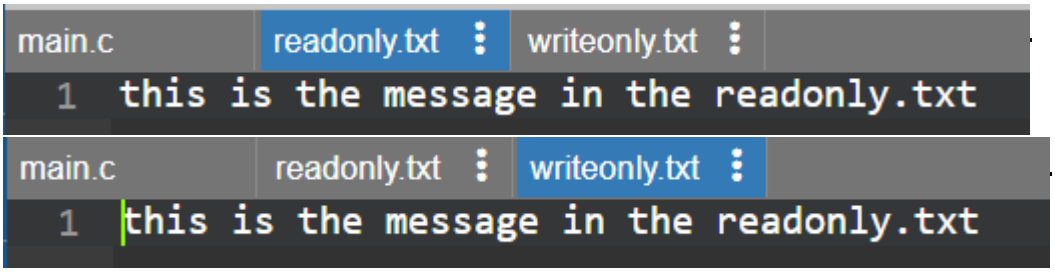
1.Code

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

#define BUFFER_SIZE 1024

int main() {
    int fd_source, fd_dest;
    char *source_path = "readonly.txt";
    char *dest_path = "writeonly.txt";
    char buffer[BUFFER_SIZE];
    ssize_t bytes_read, bytes_written;

    fd_source = open(source_path, O_RDONLY);
    if (fd_source == -1) {
        perror("Cannot open source file");
        return 1;
    }

    fd_dest = open(dest_path, O_WRONLY | O_CREAT | O_TRUNC, 0644);
    if (fd_dest == -1) {
        perror("Cannot open destination file");
        close(fd_source);
```

```
    return 1;
  }

  while ((bytes_read = read(fd_source, buffer, sizeof(buffer))) > 0) {
    bytes_written = write(fd_dest, buffer, bytes_read);
    if (bytes_written != bytes_read) {
      perror("Cannot copy");
      close(fd_dest);
      close(fd_source);
      return 1;
    }
  }

  if (bytes_read == -1) {
    perror("Cannot read source file");
    close(fd_dest);
    close(fd_source);
    return 1;
  }

  printf("File copied successfully\n");
  close(fd_dest);
  close(fd_source);
  return 0;
}
```

Terminal:

```
File copied successfully
```



2.Fork() and Wait():

Code:

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

int main() {
    pid_t pid;

    // Create a new process
    pid = fork();

    if (pid < 0) {
        // Fork failed
        perror("fork");
        exit(EXIT_FAILURE);
    }

    if (pid == 0) {
        // This is the child process
        printf("Child process (PID: %d)\n", getpid());

        // Replace the child process image with a new program
        execlp("ls", "ls", NULL);

        // If exec fails
        perror("execlp");
```

```
        exit(EXIT_FAILURE);
    } else {
        // This is the parent process
        printf("Parent process (PID: %d)\n", getpid());

        // Wait for the child process to terminate
        int status;
        waitpid(pid, &status, 0);

        if (WIFEXITED(status)) {
            printf("Child process terminated with exit status %d\n", WEXITSTATUS(status));
        } else {
            printf("Child process terminated abnormally\n");
        }
    }

    return 0;
}
```

Output:

```
Parent process (PID: 23881)
Child process (PID: 23885)
a.out  main.c
Child process terminated with exit status 0
```

Conclusion : Successfully implemented system calls functions

Post Lab Descriptive Questions

1) Describe System Call Interface.

Ans: The **System Call Interface (SCI)** is the mechanism through which user-space applications interact with the operating system's kernel to request services or perform operations that require privileged access to hardware or system resources.

Key Concepts of the System Call Interface

1. **User Space vs. Kernel Space:**
   - **User Space:** This is where user applications run. Applications in user space have limited access to hardware and system resources to protect the overall system's stability and security.
   - **Kernel Space:** This is where the core of the operating system operates, managing hardware, memory, processes, and other system resources. The kernel has unrestricted access to the hardware.
2. **Purpose of System Calls:**
   - Applications in user space often need to perform operations that they cannot directly do because of the restricted access. For example, reading from or writing to a file, allocating memory, or communicating with hardware devices.
   - To perform these operations, applications make **system calls**, which are requests for the kernel to perform a specific operation on behalf of the application.
3. **System Call Process:**
   - **Invocation:** An application issues a system call by using a library function (e.g., open(), read(), write()) that interacts with the operating system.
   - **Context Switch:** The system call triggers a context switch from user mode to kernel mode. The CPU switches to a higher privilege level where it can execute the system call code inside the kernel.
   - **Execution:** The kernel performs the requested operation (e.g., reading data from a disk, sending data over a network).
   - **Return:** After completing the operation, the kernel switches back to user mode and returns the result of the system call to the application.

2) List the types of System Calls.

Ans: **1. Process Control**

These system calls manage the processes in an operating system. They allow for

process creation, termination, execution, and management.

- **Examples:**
  - fork(): Creates a new process by duplicating the existing process.
  - exec(): Replaces the current process image with a new process image.
  - exit(): Terminates the calling process.
  - wait(): Makes a process wait until its child processes terminate.
  - getpid(): Returns the process ID of the calling process.
  - kill(): Sends a signal to a process, often used to terminate a process.

2. File Management

These system calls handle the operations on files, such as creating, deleting, reading, writing, and manipulating files and directories.

- **Examples:**
  - open(): Opens a file for reading, writing, or both.
  - close(): Closes an open file descriptor.
  - read(): Reads data from a file into a buffer.
  - write(): Writes data from a buffer to a file.
  - lseek(): Repositions the file offset of an open file.
  - unlink(): Deletes a file.
  - stat(): Retrieves information about a file.

3. Device Management

These system calls manage device operations, allowing user programs to request and release devices, and perform operations on devices.

- **Examples:**
  - ioctl(): Performs a device-specific operation on a file descriptor.
  - read(): Reads data from a device (often through file descriptors).
  - write(): Writes data to a device.
  - open(): Opens a device (often represented as a file in Unix-like systems).
  - close(): Closes a device.

4. Information Maintenance

These system calls are used to get or set information about processes, files, devices, or the system itself.

- **Examples:**
  - getpid(): Gets the process ID of the calling process.
  - gettimeofday(): Retrieves the current time.
  - getuid(): Gets the real user ID of the calling process.
  - setuid(): Sets the user ID of the calling process.

- gethostname(): Retrieves the name of the current host.
- uname(): Retrieves information about the system (e.g., kernel version).