LAB FILE ASSIGNMENT (Operating System)

Name : Gaurav Bhujel

Roll No. : (PAS078BEI014)

Question: Describe and discuss the use case of following:

1. *fork()*

The fork() system call is a fundamental operation in operating systems. It is used to create a new process by duplicating the calling process. The new process is called the child process, while the calling process is referred to as the parent process.

- **Use Case**: fork() is used to create a new process, which runs concurrently with the parent process. After a new child process is created, both processes will execute the next instruction following the fork() system call. This is essential for tasks that require concurrent processing, such as handling multiple client requests in a server application.

2. *exec()*

The exec() family of functions replaces the current process image with a new process image. This means that when you use the exec command, the current process terminates, and a new process starts running the specified executable.

- **Use Case**: exec() is used to run an executable file in the context of an already existing process. This is commonly used in scenarios where a shell or parent process needs to run a different program, replacing itself entirely with the new program.

3. *getpid()*

The getpid() function returns the process ID (PID) of the calling process.

- **Use Case**: When a parent process creates a child process using fork(), it can use getpid() to get its own PID and getppid() to get the parent

process ID. This is useful for managing and coordinating between the parent and child processes, especially in debugging and process control.

4. *wait()*

The wait() function is used by a parent process to wait for its child processes to terminate. When a parent process calls wait(), it pauses execution until one of its child processes exits or a signal is received.

- **Use Case**: In scenarios where a parent process spawns multiple child processes, wait() helps manage the lifecycle of each child process by waiting for their termination one by one. This ensures proper cleanup and synchronization of resources.

5. *stat()*

The stat() function retrieves information about a file or directory. It provides details such as the file's size, permissions, owner, and timestamps.

- **Use Case**: stat() is commonly used in file management and system programming to obtain detailed information about files and directories. This information is crucial for tasks such as file system navigation, access control, and data integrity checks.

6. *opendir()*

The opendir() function is used to open a directory stream. This stream can then be used to read the contents of the directory using functions like readdir() and to close the directory stream with closedir().

- **Use Case**: opendir() is used in utilities that need to manage or analyze directory contents, such as backup tools, file managers, or custom scripts. It provides a way to iterate through the contents of a directory.

7. *readdir()*

The readdir() function reads directory entries from a directory stream opened with opendir(). It allows iteration through the contents of a directory and retrieves information about each file or subdirectory within it.

- **Use Case**: readdir() is used to list all entries in a directory, which is useful for file management applications, directory traversal scripts, and system utilities that need to process files within a directory.

8. *close()*

The close() function is used to close a file descriptor, which is an integer handle used by the operating system to identify an open file, socket, or other I/O resource. Closing a file descriptor releases the associated resources and marks it as no longer in use.

- **Use Case**: close() is essential for resource management in applications. It ensures that file descriptors are properly closed after their use, preventing resource leaks and allowing the system to reuse those resources for other processes.

// C program for producer consumer problem

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

// Initialize a mutex to 1
int mutex = 1;

// Number of full slots as 0
int full = 0;
```

```
// Number of empty slots as the size of the buffer
int empty = 10, x = 0;

// Function to produce an item and add it to the buffer
void producer() {
    --mutex;  // Decrease mutex by 1
    ++full;   // Increase number of full slots by 1
    --empty;  // Decrease number of empty slots
    x++;
    printf("\nProducer produces item %d", x);
    ++mutex;  // Increase mutex by 1
}

// Function to consume an item and remove it from the
buffer
void consumer() {
    --mutex;  // Decrease mutex by 1
    --full;   // Decrease number of full slots by 1
    ++empty;  // Increase number of empty slots
    printf("\nConsumer consumes item %d", x);
    x--;
    ++mutex;  // Increase mutex by 1
```

```
}

int main() {
    int n;

    while (1) {
        printf("\n\n1. Press 1 for Producer");
        printf("\n2. Press 2 for Consumer");
        printf("\n3. Press 3 for Exit");

        printf("\nEnter your choice: ");
        scanf("%d", &n);

        // Switch cases
        switch (n) {
            case 1:
                if ((mutex == 1) && (empty != 0)) {
                    producer();
                } else {
                    printf("Buffer is full");
                }
                break;
```

```
        case 2:
            if ((mutex == 1) && (full != 0)) {
                consumer();
            } else {
                printf("Buffer is empty!");
            }
            break;

        case 3:
            exit(0);

        default:
            printf("Invalid choice");
            break;
      }
   }

   return 0;
}
```