OS Assignment

Submitted by:- Aagaman K.C.(pas078bei001)

Class Roll No:- 01

Detailed Discussion on System Calls and Functions

1. fork()

The `fork()` system call is a fundamental operation in Unix-based operating systems. It creates 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. After a new child process is created, both processes will execute the next instruction following the `fork()` system call.

- **Use Case**: `fork()` is used in scenarios where a process needs to create a copy of itself to perform tasks concurrently. For example, in a web server, each incoming request can be handled by a new child process created using `fork()`, allowing multiple requests to be processed simultaneously.

2. exec()

The `exec()` family of functions replaces the current process image with a new process image. It runs an executable file in the context of an already existing process, replacing the previous executable.

- **Use Case**: `exec()` is used when a process needs to execute a different program. For example, a shell process may use `exec()` to replace itself with a new program specified by the user, such as running a script or an executable binary.

3. getpid()

The `getpid()` function returns the process ID (PID) of the calling process. It is simple to use and does not take any arguments.

- **Use Case**: `getpid()` is commonly used in scenarios where a process needs to know its own PID, such as for logging, debugging, or managing child processes created using `fork()`. For example, a parent process may log its PID and the PID of its child processes for monitoring purposes.

4. wait()

The `wait()` function makes a parent process 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**: `wait()` is used to synchronize the termination of child processes. In a server application, a parent process may create multiple child processes to handle client requests and use `wait()` to ensure that it cleans up resources properly after each child process terminates.

5. stat()

The `stat()` function retrieves information about a file or directory, such as its size, permissions, owner, and timestamps. This information is stored in a structure.

- **Use Case**: `stat()` is widely used in file management and system programming. For example, backup tools use `stat()` to gather metadata about files before copying them to ensure that all file attributes are preserved.

6. opendir()

The `opendir()` function opens a directory stream, which can then be used to read the contents of the directory with functions like `readdir()` and to close the directory stream with `closedir()`.

- **Use Case**: `opendir()` is commonly used in utilities that need to manage or analyze directory contents, such as file managers, custom scripts, or backup tools. For example, a program may use `opendir()` to open a directory and then use `readdir()` to iterate through its contents.

7. readdir()

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

- **Use Case**: `readdir()` is used in programs that need to process all files in a directory. For example, a file search utility may use `readdir()` to list all files in a directory and then search each file for a specific pattern.

8. close()

The `close()` function closes 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 in resource management to ensure that file descriptors are properly closed after their use. For example, after reading from or writing to a file, a program should call `close()` to free the file descriptor and avoid resource leaks.

Corrected and Improved C Program for Producer-Consumer Problem

```
c
Copy code
#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;

// Item count
int x = 0;

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

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

int main() {
    int n;
    while (1) {
        printf("\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);
            break;

        default:
            printf("Invalid choice. Please try again.");
        }
    }

    return 0;
}
```

This refined program ensures proper synchronization between the producer and consumer functions using mutex and handles the edge cases where the buffer is full or empty.