

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.

Fork system call is used for creating a new process, which is called child 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.

2.exec()

exec is a functionality of an operating system that runs an executable file in the context of an already existing process, replacing the previous executable.

It is used to replace the current shell process with a new process. This means that when you use the 'exec' command, the current shell process terminates, and a new process starts.

3.getpid()

The getpid() function is simple to use and does not take any arguments. It returns the PID of the process that calls it.

When a parent process creates a child process using `fork()`, it can use `getpid()` and `getppid()` which gets the parent process ID) to manage and coordinate between the two processes.

4.wait()

The Wait() function in operating systems 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.

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.

5.stat()

The stat() function in operating systems is used to retrieve information about a file or directory. It provides details such as the file's size, permissions, owner, and timestamps. This function is commonly used in file management and system programming.

The stat() function retrieves the status of a file and stores it in a structure, which contains various fields representing different attributes of the file.

6.opendir()

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

`opendir()` is commonly used in utilities that need to manage or analyze directory contents, such as backup tools, file managers, or custom scripts.

7. `readdir()`

The `readdir()` function in operating systems is used to read directory entries from a directory stream opened with `opendir()`. It allows you to iterate through the contents of a directory and retrieve information about each file or subdirectory within it.

8. `close()`

The `close()` function in operating systems 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.

```
// 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 size
```

```
// of buffer
```

```
int empty = 10, x = 0;
```

```
// Function to produce an item and
```

```
// add it to the buffer
```

```
void producer()
```

```
{
```

```
    // Decrease mutex value by 1
```

```
    --mutex;
```

```
    // Increase the number of full
```

```
    // slots by 1
```

```
    ++full;
```

```
    // Decrease the number of empty
```

```
    // slots by 1
```

```
    --empty;
```

```
    // Item produced
```

```
    x++;
```

```
    printf("\nProducer produces"
```

```
        "item %d",
```

```
        x);
```

```
    // Increase mutex value by 1
```

```
    ++mutex;
```

```
}
```

```
// Function to consume an item and
```

```

// remove it from buffer

void consumer()
{
    // Decrease mutex value by 1
    --mutex;

    // Decrease the number of full
    // slots by 1
    --full;

    // Increase the number of empty
    // slots by 1
    ++empty;
    printf("\nConsumer consumes "
           "item %d",
           x);
    x--;

    // Increase mutex value by 1
    ++mutex;
}

int main()
{
    int n, i;

    printf("\n1. Press 1 for Producer"

```

```
"\n2. Press 2 for Consumer"
```

```
"\n3. Press 3 for Exit");
```

```
// Using '#pragma omp parallel for'
```

```
// can give wrong value due to
```

```
// synchronization issues.
```

```
// 'critical' specifies that code is
```

```
// executed by only one thread at a
```

```
// time i.e., only one thread enters
```

```
// the critical section at a given time
```

```
#pragma omp critical
```

```
for (i = 1; i > 0; i++) {
```

```
    printf("\nEnter your choice:");
```

```
    scanf("%d", &n);
```

```
// Switch Cases
```

```
switch (n) {
```

```
case 1:
```

```
    // If mutex is 1 and empty
```

```
    // is non-zero, then it is
```

```
    // possible to produce
```

```
    if ((mutex == 1)
```

```
    && (empty != 0)) {  
        producer();  
    }
```

```
    // Otherwise, print buffer  
    // is full  
    else {  
        printf("Buffer is full!");  
    }  
    break;
```

case 2:

```
    // If mutex is 1 and full  
    // is non-zero, then it is  
    // possible to consume  
    if ((mutex == 1)  
        && (full != 0)) {  
        consumer();  
    }
```

```
    // Otherwise, print Buffer  
    // is empty  
    else {  
        printf("Buffer is empty!");  
    }
```

```
break;
```

```
// Exit Condition
```

```
case 3:
```

```
    exit(0);
```

```
    break;
```

```
}
```

```
}
```

```
}
```