**How to create a child process with fork in Linux**

The "fork" system call is a fundamental feature of operating systems that allows a process to create a new process, known as a child process. The fork system call essentially creates an exact copy of the existing process, including its memory, file descriptors, and other attributes.

When a fork system call is executed, the operating system creates a new process by duplicating the existing process. The new process, called the child process, starts execution at the same point as the parent process, known as the fork point. After the fork, the parent process and the child process are separate and independent, with their own copies of resources.

The fork system call is commonly used in operating systems to implement features such as multitasking and process creation. By forking a process, an operating system can create multiple processes that run concurrently, each executing different tasks.

Here's a basic example of how the fork system call works in C:

#include <stdio.h>

#include <unistd.h>

int main() {

 pid\_t pid = fork();

 if (pid == -1) {

 // Error occurred

 perror("fork");

 return 1;

 } else if (pid == 0) {

 // Child process

 printf("Hello from child process!\n");

 } else {

 // Parent process

 printf("Hello from parent process!\n");

 }

 return 0;

}

In this example, when the program is executed, the **fork()** call is made. If the **fork()** call returns -1, an error occurred. If it returns 0, it means the process is the child process. If it returns a positive value, it means the process is the parent process, and the returned value is the process ID (PID) of the child process.

The program then prints a different message depending on whether it is the parent or child process. Both the parent and child processes continue executing from the point of the **fork()** call, but they are separate processes with their own copies of variables and resources.

Certainly! Here are a few more examples that demonstrate the usage of the **fork()** system call:

Example 1: Creating Multiple Child Processes

 #include <stdio.h>

#include <unistd.h>

int main() {

 for (int i = 0; i < 3; i++) {

 pid\_t pid = fork();

 if (pid == -1) {

 // Error occurred

 perror("fork");

 return 1;

 } else if (pid == 0) {

 // Child process

 printf("Hello from child process with ID %d\n", getpid());

 return 0;

 }

 }

 // Parent process

 printf("Hello from parent process with ID %d\n", getpid());

 return 0;

}

In this example, the parent process creates three child processes using a **for** loop. Each child process prints its own process ID (**getpid()**), and the parent process also prints its own process ID. The output will show multiple child processes being created and executing concurrently.

Example 2: Executing Different Code in Parent and Child Processes

#include <stdio.h>

#include <unistd.h>

int main() {

 pid\_t pid = fork();

 if (pid == -1) {

 // Error occurred

 perror("fork");

 return 1;

 } else if (pid == 0) {

 // Child process

 printf("Hello from child process!\n");

 execlp("/bin/ls", "ls", "-l", NULL);

 // If execlp() is successful, the child process will execute the "ls" command

 // If execlp() fails, an error will be returned

 perror("execlp");

 } else {

 // Parent process

 printf("Hello from parent process!\n");

 }

 return 0;

}

In this example, the child process uses the **execlp()** function to execute the "ls" command, which lists the files and directories in the current directory. The parent process simply prints a message. When the program is executed, the output will show the message from the parent process, followed by the output of the "ls" command executed by the child process.

These examples demonstrate different ways in which the **fork()** system call can be used to create child processes and execute separate code paths in the parent and child processes.