

Laboratory 1

Title of the Laboratory Exercise: Programs using process management system calls

1. Introduction and Purpose of Experiment

A system call is a programmatic way in which a computer program requests a service from the kernel of the operating system it is executed on. There are different types of system calls developed for various purposes. They are mainly classified as process management, file management, directory management. By solving the problems students will be able to apply process management system calls

Aim and Objectives

Aim

- To develop programs involving process management system calls

Objectives

At the end of this lab, the student will be able to

- Use different process management system calls
- Apply different system calls wherever required
- Create C programs using process management system calls

2. Experimental Procedure

- i. Analyse the problem statement
- ii. Design an algorithm for the given problem statement and develop a flowchart/pseudo-code
- iii. Implement the algorithm in C language
- iv. Compile the C program
- v. Test the implemented program
- vi. Document the Results
- vii. Analyse and discuss the outcomes of your experiment

3. Questions

Implement the following operations in C

Create four different processes (with different process ID) and assign four different tasks (addition, subtraction, Multiplication, division) to each process. All processes should display the result along with its process ID and parent process ID.

4. Calculations/Computations/Algorithms

1. If (pid = fork())
2. do addition operation
3. kill process
4. else
5. do subtraction operation
6. If (pid = fork())
7. do multiplication
8. else
9. do division

5. Presentation of Results

main.cpp

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

void print_proc_details() {
    std::cout << "Process id : [ " << getpid() << " ]" << std::endl;
    std::cout << "Parent Process id : [ " << getppid() << " ]" << std::endl;
}

int main(int, char**) {
    long long a, b, res;

    // print_proc_details();

    std::cout << "Enter two numbers : ";
    std::cin >> a >> b;

    pid_t pid;

    if (pid = fork()) {
        // parent process
        // do addition here
        print_proc_details();
        res = a + b;
        std::cout << a << " + " << b << " = " << res << std::endl;
        std::cout << std::endl;
    }
```

```
    exit(EXIT_SUCCESS);
} else {
    // child process
    // do subtraction here
    print_proc_details();
    res = a - b;
    std::cout << a << " - " << b << " = " << res << std::endl;
    std::cout << std::endl;

    if(pid = fork()) {
        // the child itself
        if (fork() == 0) {
            print_proc_details();
            res = a * b;
            std::cout << a << " * " << b << " = " << res << std::endl;
            std::cout << std::endl;

            exit(EXIT_SUCCESS);
        }
    } else {
        // child's child
        print_proc_details();
        res = a / b;
        std::cout << a << " / " << b << " = " << res << std::endl;
        std::cout << std::endl;

        exit(EXIT_SUCCESS);
    }

    exit(EXIT_SUCCESS);
}
}
```

OUTPUT:

```
shadowleaf@SHADOWLEAF-ROG /mnt/d/University-Work/University-Work-SEM-05/OS-Lab/Lab01
/mnt/d/University-Work/University-Work-SEM-05/OS-Lab/Lab01/build/Lab01
Enter two numbers : 8 4
Process id : [ 5610 ]
Parent Process id : [ 5215 ]
8 + 4 = 12

Process id : [ 5611 ]
Parent Process id : [ 1 ]
8 - 4 = 4

Process id : [ 5612 ]
Parent Process id : [ 5611 ]
8 / 4 = 2

Process id : [ 5613 ]
Parent Process id : [ 5611 ]
8 * 4 = 32
```

Explanation:

The program starts its execution with its main process, to create a new process, `fork()` is called, now this duplicates the current process, now the program has 2 processes, one of these, the main process is used to do addition operation, the child process does subtraction, the respective process id's are printed on the console, namely 5610 and 5611, the main process is killed using `exit()` function, and the child now calls `fork()` to create another process, the child is killed using `exit()` and the process is repeated to for multiplication and division operation.

6. Analysis and Discussions

fork - creates a new process

prototype: `#include <unistd.h> pid_t fork(void);`

Description: The *fork()* function shall create a new process. The new process (child process) shall be an exact copy of the calling process (parent process), some of its properties are:

1. The child process shall have a unique process ID.
2. The child process shall have a different parent process ID, which shall be the process ID of the calling process.
3. The child process shall have its own copy of the parent's message catalog descriptors.
4. A process shall be created with a single thread. If a multi-threaded process calls *fork()*, the new process shall contain a replica of the calling thread and its entire address space, possibly including the states of mutexes and other resources. Consequently, to avoid errors, the child process may only execute async-signal-safe operations until such time as one of the *exec* functions is called. Forkhandlers may be established by means of the *pthread_atfork()* function in order to maintain application invariants across *fork()* calls.
5. The child does not inherit its parent's memory locks

After *fork()*, both the parent and the child processes shall be capable of executing independently before either one terminates.

return value: Upon successful completion, *fork()* shall return 0 to the child process and shall return the process ID of the child process to the parent process. Both processes shall continue to execute from the *fork()* function. Otherwise, -1 shall be returned to the parent process, no child process shall be created, and *errno* shall be set to indicate the error.

associated functions:

`waitpid()`: This function takes a numeric argument, which is the process-id to wait for. The return value is that of the `waitpid(2)` system call.

`wait()`: This function waits for the first child to die. The return value is that of the `wait(2)` system call.

2. Conclusions

`fork()` is a system call that is used to create processes, a process in simple terms can be defined as a thread under execution.

- If **`fork()`** returns a negative value, the creation of a child process was unsuccessful.
- **`fork()`** returns a zero to the newly created child process.
- **`fork()`** returns a positive value, the ***process ID*** of the child process, to the parent. The returned process ID is of type **`pid_t`** defined in **`sys/types.h`**. Normally, the process ID is an integer. Moreover, a process can use function **`getpid()`** to retrieve the process ID assigned to this process.

In UNIX it will make an exact copy of the parent's address space and give it to the child. Therefore, the parent and child processes have separate address spaces.

`fork()` can fail for the following reasons:

- **`fork()`** failed to allocate the necessary kernel structures because memory is tight.
- **`fork()`** is not supported on this platform (for example, hardware without a Memory-Management Unit).

Note: Under Linux, **`fork()`** is implemented using copy-on-write pages, so the only penalty that it incurs is the time and memory required to duplicate the parent's page tables, and to create a unique task structure for the child.

3. Comments

1. Limitations of Experiments

fork() will fail if it cannot allocate sufficient memory to copy the parent's page tables and allocate a task structure for the child.

2. Limitations of Results

Due to the fact that **fork()** can fail at certain situations, error handling is not done by the program.

3. Learning happened

The concept of process management and creation using fork system call was learnt in this lab.

4. Recommendations

The errors given by **fork()** can be caught and necessary countermeasures should be taken.