Exec () System call

Objective:

This lab describes how a program can execute, wait, terminate, and control children processes using system calls.

Activity Outcomes:

On completion of this lab students will be able to:

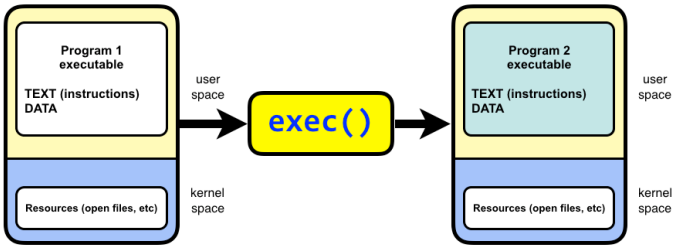
- Write programs that uses process creation system call fork ()
- Use exec system call

Exec () System call – a family of six functions

The `exec()` system call is part of a family of functions in UNIX like operating systems (such as Linux) that are used to replace the current process image with a new process image. Essentially, when a process calls an `exec()` function, the current process is replaced with a new program, and the process ID (PID) remains the same. The new program starts executing from its entry point, and everything in the old program's memory is lost.

The exec family of system calls

The exec family of system calls replaces the program executed by a process. When a process calls exec, all code (text) and data in the process is lost and replaced with the executable of the new program. Although all data is replaced, all open file descriptors remains open after calling `exec` unless explicitly set to close-on-exec. In the below diagram a process is executing Program 1. The program calls `exec` to replace the program executed by the process to Program 2.



`exec()` family consists of six different functions, each designed to execute a new program, with slight differences in how they handle arguments and environment variables.

1. execl()

Syntax:

int execl(const char *path, const char *arg, ..., NULL);

The `execl()` function takes a variable number of arguments.

Each argument is passed to the new program as separate parameters.

Path: Path to the executable.

arg: The first argument should be the name of the program (conventionally).

Other arguments follow and must end with a `NULL` pointer.

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

int main(int argc, char *argv[]){
    printf("This is exec1.c, PID of exec1.c is : %d\n", getpid());
    execl("./ex2", "ex2", NULL);
    printf("Back to execl1.c (this won't be printed if execl is successful)\n");
    return 0;
}
```

2. execv()

Syntax:

int execv(const char *path, char *const argv[]);

This function is similar to `execv()`, but the arguments are passed as an array rather than a variable number of arguments.

argv: An array of strings representing the command line arguments.

example.c

CODE:

```c
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
int main(int argc, char *argv[])
{
    printf("PID of example.c = %d\n", getpid());
    char *args[] = {"Hello", "C", "Programming", NULL};
    execv("./hello", args);
    printf("Back to example.c");
    return 0;
}
```

hello.c

```
ubuntu@ubuntu:~/Documents$ gcc -o example example.c
ubuntu@ubuntu:~/Documents$ gcc -o hello hello.c
ubuntu@ubuntu:~/Documents$ ./example
PID of example.c = 4733
We are in Hello.c
PID of hello.c = 4733
ubuntu@ubuntu:~/Documents$
```

3.execle()

Syntax:

int execle(const char *path, const char *arg, ..., NULL, char *const envp[]);

`execle()` works like `execl()`, but also allows you to specify a custom environment for the new process.

envp: An array of strings representing the environment variables for the new process.

4.

```c
#include <stdio.h>
#include <unistd.h>  // For execle()

int main() {
    printf("Before execle()\n");
    execle("/bin/ls", "ls", "-l", NULL, NULL);
    // If execle() fails
    perror("execle failed");
    return 1;
```

execve()

Syntax:

int execve(const char *path, char *const argv[], char *const envp[]);

`execve()` is the most fundamental `exec()` function and is the system call that all other `exec()` functions internally use.

It takes the path to the executable, the array of arguments, and an array of environment variables.

5.

```c
#include <stdio.h>
#include <unistd.h>  // For execve()

int main() {
    printf("Before execve()\n");
    char *args[] = {"ls", "-l", NULL};  // Arguments for execve()
    char *envp[] = {NULL};  // Environment variables for execve()
    execve("/bin/ls", args, envp);
    // If execve() fails
    perror("execve failed");
    return 1;
}
```

execlp()

Syntax:

int execlp(const char *file, const char *arg, ..., NULL);

The `execlp()` function behaves like `execl()`, but instead of specifying the full path to the executable, the system searches for the executable in the directories listed in the `PATH` environment variable.

```c
#include <stdio.h>
#include <unistd.h>  // For execlp()

int main() {
    printf("Before execlp()\n");
    execlp("ls", "ls", "-l", NULL);
```

6. execvp()

Syntax:

int execvp(const char *file, char *const argv[]);

Similar to `execv()`, `execvp()` takes the arguments as an array and looks for the executable in the `PATH` environment variable.

The `exec()` system call does not create a new process; it simply replaces the current process image with a new program.

After an `exec()` call, the original program ceases to exist, and the new program runs in the same process.

The PID of the process remains unchanged.

If the `exec()` call is successful, it does not return. If it fails (e.g., if the executable is not found), the function returns `1` and sets `errno` to indicate the error.

```c
#include <stdio.h>
#include <unistd.h>  // For execvp()

int main() {
    printf("Before execvp()\n");
    char *args[] = {"ls", "-l", NULL};
    execvp("ls", args);
    // If execvp() fails
    perror("execvp failed");
    return 1;
}
```

Example (using `execlp()`):

```c
#include <stdio.h>
#include <unistd.h>

int main() {
    printf("Before exec()\n");

    // Replace current process with /bin/ls
    execlp("ls", "ls", "-l", NULL);

    // This will not be printed if exec() is successful
    printf("This line will not be printed\n");

    return 0;
}
```

If successful, the `execlp()` call in this example will replace the current process with the `ls` command, listing the directory contents. The process does not return to the original program.

Summary of Differences:

Function	Arguments (Command)	Arguments (Env Variables)	Path Search
`execl`	Variable list	No	No
`execv`	Array	No	No
`execle`	Variable list	Yes	No
`execve`	Array	Yes	No

| `execlp` | Variable list | No | Yes |
| `execvp` | Array | No | Yes |

This family of functions provides flexibility depending on how you want to pass arguments and environment variables to the new process.

Some bulletpoint guide on how to check and handle errors, ensure clean process termination, and interpret status codes for the `fork()`, `exec()`, `wait()`, and `exit()` system calls, without including code:

Error Handling

1. **Fork System Call (`fork()`):**

   Check Return Value:

   If `fork()` returns a negative value, it indicates an error in creating a new process.

   Handle Success and Failure:

   If `fork()` returns `0`, you are in the child process.

   If `fork()` returns a positive value, you are in the parent process, and the return value is the child's process ID.

2. **Exec System Call (`exec()`):**

   Check Return Value:

   `exec()` functions replace the current process image; they do not return on success.

   if `exec()` fails, it returns `1`, and an error message can be retrieved.

3. **Wait System Call (`wait()` or `waitpid()`):**

   Check Return Value:

If `wait()` or `waitpid()` returns `1`, it indicates an error in waiting for the child process.

Handle Waiting Errors:

Verify that the process ID passed to `wait()` matches the child process.

4. **Exit System Call (`exit()`):**

No Direct Return Value:

`exit()` terminates the process and does not return a value. Ensure that `exit()` is called in all scenarios where the process needs to terminate.

Process Termination

1. **Ensure Proper Exit in Child Process:**

Use `exit()`:

The child process should call `exit()` to terminate properly.

2. **Ensure Proper Exit in Parent Process:**

Wait for Child:

The parent process should wait for the child to complete using `wait()` or `waitpid()` to avoid premature termination and zombie processes.

3. **Handle Zombie Processes:**

Ensure Reaping:

Always use `wait()` or `waitpid()` to handle the child process's exit to prevent zombie processes.

Status Codes

1. **Check Exit Status:**

Use `WIFEXITED(status)`:

Check if the child process terminated normally.

2. **Retrieve Exit Status:**

   Use `WEXITSTATUS(status)`:

   Extract the exit status of the child process if it terminated normally.

3. **Check for Abnormal Termination:**

   Use `WIFSIGNALED(status)`:

   Determine if the child process was terminated by a signal.

4. **Check for Core Dumps:**

   Use `WCOREDUMP(status)`:

   Check if the child process generated a core dump.

I've added the references links for more details:

https://www2.it.uu.se/education/course/homepage/os/vt18/module2/exec/

Graded Task:

In this activity, you are required to perform tasks given below:

1) Print something and Check id of the parent process **[Estimated 5-10 mins]**
2) Create a child process and print child process id in parent process. **[Estimated 10 mins]**
3) Create a process and make it an orphan. **[Estimated 5 mins]**
   Hint: To, illustrate this insert a sleep statement into the child's

code. This ensured that the parent process terminated before its child.

4) Write a two programs one is ex1 and second is ex2 and then pass an argument as an age variable from ex1 to ex2 using exec sysetm call. **[Estimated 10 mins]**

5) Write a C++ program that creates an array of size 1000 and populates it with random integers between 1 and 100. Now, it creates two child processes. The first child process finds how many prime numbers are there among first 500 number while the second child process finds the number of prime numbers among the remaining 500 numbers. **[Estimated 20 mins]**

6) Write a program that sets a custom environment variable (e.g., MY_VAR="HelloWorld") and then uses execle to execute /usr/bin/env (a command that prints all environment variables).

[Estimated 5 mins]

7) Write a program that prompts the user to input a command, and then uses execvp to execute the input command dynamically. Example: If the user inputs ps, the program will run execvp("ps", args);.

[Estimated 10 mins]