Assignment No. 2

Tazmeen Afroz
Roll No: 22P-9252

October 24, 2024

Contents

1 Introduction

The ‘exec‘ family of system calls plays a crucial role in how modern operating systems execute programs. These system calls replace the current process image with a new process image, enabling seamless process execution. In this document, we will cover the different variants of the ‘exec‘ system calls and their specific use cases in operating systems.

2 Usage and Purpose

Each of the ‘exec‘ system calls is designed for slightly different situations, such as whether arguments are known at compile time or runtime, whether environment variables need to be passed, or whether the program is located within the system ‘PATH‘. These system calls are particularly useful for:

- Running external programs from within a C program.
- Creating processes that replace themselves with another program.
- Passing specific environment variables to the new program.
- Efficient process management in multi-tasking environments.

3 Comparison of Exec Variants

Feature	**execvp**	**execv**	**execve**
PATH search	Yes	No	No
Full path needed	No	Yes	Yes
Array arguments	Yes	Yes	Yes
Environment control	No	No	Yes

4 Base Program

Before diving into different ‘exec‘ system calls, here is a base program that we will use for testing. The program calculates the sum of numbers passed via command-line arguments.

4.1 Code Implementation

```
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 int main(int argc, char *argv[]) {
5     int sum = 0;
6     for(int i = 1; i < argc; i++) {
7         sum += atoi(argv[i]);
```

```
    }
    printf("Sum: %d\n", sum);
    return 0;
}
```

4.2 Explanation

This base program:

- Takes command-line arguments as input.
- Converts string arguments to integers using `atoi()`.
- Calculates their sum and prints it.
- Example: If called with arguments ”10”, ”20”, and ”30”, it outputs 60.

5 The ‘execl‘ System Call

5.1 Purpose

The ‘execl()‘ system call is used when:

- Arguments are known at compile time.
- The full path to the executable is known.
- Arguments are passed individually as separate parameters.

5.2 Implementation

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

int main() {
    pid_t pid = fork();

    if (pid < 0) {
        perror("Fork failed");
        exit(1);
    }

    if (pid == 0) {  // Child process
        printf("Child process executing execl\n");
        execl("./sum", "sum", "10", "20", "30", NULL);
        perror("execl failed");
        exit(1);
```

```
    } else {  // Parent process
        wait(NULL);
        printf("execl example completed\n");
    }
}
```

5.3 Explanation

The ‘execl()‘ call is designed for situations where you know the arguments and the path to the executable at compile time. In this case, the ‘sum‘ program is executed with the arguments ”10”, ”20”, and ”30”. A child process is created using ‘fork()‘, and the child process replaces itself with the ‘sum‘ program using ‘execl()‘. If the ‘execl()‘ call fails, an error message is printed.

5.4 Practical Scenario in OS

In operating systems, ‘execl()‘ can be used for launching specific applications from within another program. For instance, consider a simple system utility that launches a calculator app. The app path and arguments (if any) are known at compile time.
Scenario: Opening a text editor from a terminal using ‘execl()‘, where the editor path and file to be opened are predefined.

6 The ‘execlp‘ System Call

6.1 Purpose

The ‘execlp()‘ system call is used when:

- The program to be executed is available in the system ‘PATH‘.
- No need to specify the full path of the program.
- Convenient for executing common system commands.

6.2 Implementation

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

int main() {
    pid_t pid = fork();

    if (pid == 0) {
        printf("Child process executing execlp\n");
```

```
11          execlp("ps", "ps", "aux", NULL);
12          perror("execlp failed");
13          exit(1);
14      } else {
15          wait(NULL);
16          printf("execlp example completed\n");
17      }
18  }
```

6.3 Explanation

In this example, the ‘execlp()‘ system call is used to execute the ‘ps aux‘ command, which lists system processes. This command is located in the system ‘PATH‘, meaning there is no need to provide the full path to the executable. The child process, created by ‘fork()‘, executes the ‘ps‘ command using ‘execlp()‘ and then exits.

6.4 Practical Scenario in OS

Scenario: Suppose a desktop environment wants to list all currently running applications in the background (similar to ‘Task Manager‘). The system can invoke ‘execlp()‘ to run the ‘ps aux‘ command to gather process information without needing the full path.
Example: Opening a system monitoring tool that runs ‘ps aux‘ to show running processes.

7 The ‘execle‘ System Call

7.1 Purpose

The ‘execle()‘ system call is used when:

- Custom environment variables need to be passed to the new process.
- The full path to the executable is known.
- Arguments are known at compile time.

7.2 Implementation

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <unistd.h>
4  #include <sys/wait.h>
5
6  int main() {
7      char *const envp[] = {"PATH=/bin:/usr/bin", NULL};
```

```
    pid_t pid = fork();

    if (pid == 0) {
        execle("./sum", "sum", "15", "25", "35", NULL, envp);
        perror("execle failed");
        exit(1);
    } else {
        wait(NULL);
    }
}
```

7.3 Explanation

Here, the ‘execle()‘ system call is used to run the ‘sum‘ program, but with custom environment variables passed in. In this example, we set the ‘PATH‘ variable to only include ‘/bin‘ and ‘/usr/bin‘, which restricts the environment in which the ‘sum‘ program will execute. The ability to customize the environment is particularly useful in scenarios where the new process needs a modified or restricted environment.

7.4 Practical Scenario in OS

Scenario: Consider launching a program that requires a specific version of a tool or library, such as launching a browser in a different environment for testing purposes with custom paths set.

8 The ‘execvp‘ System Call

8.1 Purpose

The ‘execvp()‘ system call is used when:

- The program to be executed is located in the system ‘PATH‘.
- Arguments are passed as an array.

8.2 Implementation

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

int main() {
    char *const args[] = {"ps", "aux", NULL};
    pid_t pid = fork();
```

```
9
10      if (pid == 0) {
11          execvp(args[0], args);
12          perror("execvp failed");
13          exit(1);
14      } else {
15          wait(NULL);
16      }
17  }
```

8.3 Explanation

The ‘execvp()‘ call is similar to ‘execlp()‘, except that arguments are passed as an array rather than individual parameters. In this case, the ‘ps aux‘ command is executed to list system processes.

8.4 Practical Scenario in OS

Scenario: Using ‘execvp()‘ to launch applications available in the system ‘PATH‘, such as opening a terminal emulator. This approach simplifies running well-known system commands without the need to specify the path.

9 execv System Call

9.1 Purpose

The `execv()` system call is used when:

- Full path to executable is known
- Arguments need to be passed as an array
- Flexible number of arguments needed

9.2 Implementation

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <unistd.h>
4  #include <sys/wait.h>
5
6  int main() {
7      char *const args[] = {"./sum", "40", "50", "60", NULL};
8      pid_t pid = fork();
9
10     if (pid == 0) {
11         execv(args[0], args);
12         perror("execv failed");
```

```
13          exit(1);
14      } else {
15          wait(NULL);
16      }
17      return 0;
18  }
```

9.3 Explanation

In this example, the `execv()` system call is used to execute the `sum` program with three arguments: "40", "50", and "60". The child process replaces itself with the sum program using `execv()`. The arguments are passed as an array, making it convenient when dealing with multiple arguments. The `execv()` system call is more flexible than `execl()` as it allows for a variable number of arguments to be passed easily through an array.

9.4 Practical Scenario in OS

The `execv()` system call is commonly used in operating systems when a program needs to execute another binary with a flexible set of arguments. For example, in the implementation of shell programs, where the user may execute a command with varying numbers of arguments, `execv()` can be utilized to handle these commands. The shell program forks a child process, and the child process calls `execv()` to run the desired command. This method is efficient for scripting and running programs with various command-line parameters.

10 execve System Call

10.1 Purpose

The `execve()` system call is used when:

- Full path to executable is known
- Arguments need to be passed as an array
- Custom environment variables are required

10.2 Implementation

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <unistd.h>
4  #include <sys/wait.h>
5
6  int main() {
7      char *const args[] = {"./sum", "70", "80", "90", NULL};
```

```
8       char *const envp[] = {"PATH=/bin:/usr/bin", NULL};
9       pid_t pid = fork();
10
11      if (pid == 0) {
12          execve(args[0], args, envp);
13          perror("execve failed");
14          exit(1);
15      } else {
16          wait(NULL);
17      }
18  }
```

10.3 Explanation

Here, `execve()` is used to execute the `sum` program with three arguments: "70", "80", "90" and a custom environment variable. The child process replaces itself with the sum program using `execve()`. This variant is similar to `execle()`, but it takes arguments as an array instead of individual parameters. The environment variables are also passed as an array, where each element is a string in the format "NAME=VALUE".

10.4 Practical Scenario in OS

The `execve()` system call is typically used in operating systems when a program needs to run another process with custom environment variables. It is often seen in environments where applications need to run with specific configurations, such as web servers or scripts that require specific environment paths or variables. For example, in containerized environments (like Docker), `execve()` can be used to launch processes with custom environment settings to isolate them from the host environment.

11 Outputs of 'exec' System Calls

In this section, we present the outputs for the code examples involving different 'exec' system calls.

12 Output of 'execl' System Call

The 'execl' system call replaces the current process with a new one. In our example, the base program 'sum' calculates the sum of integers passed as arguments.

12.1 Code Output

When the program is executed, the child process runs the 'sum' program with arguments "10", "20", and "30". The output is the sum of these integers.

12.2 Image of the Output

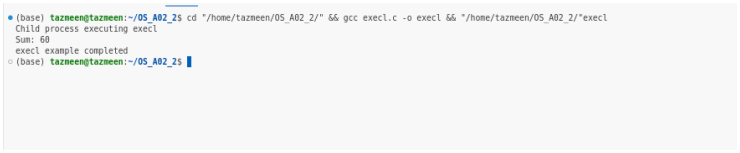

Figure 1: Output of execl system call.

13 Output of ‘execlp‘ System Call

In this example, the ‘execlp‘ system call is used to execute the ‘ps aux‘ command to display the list of running processes. Since the program is available in the system ‘PATH‘, we don’t need to specify the full path.

13.1 Code Output

When executed, the following output is displayed, showing the list of running processes:

13.2 Image of the Output

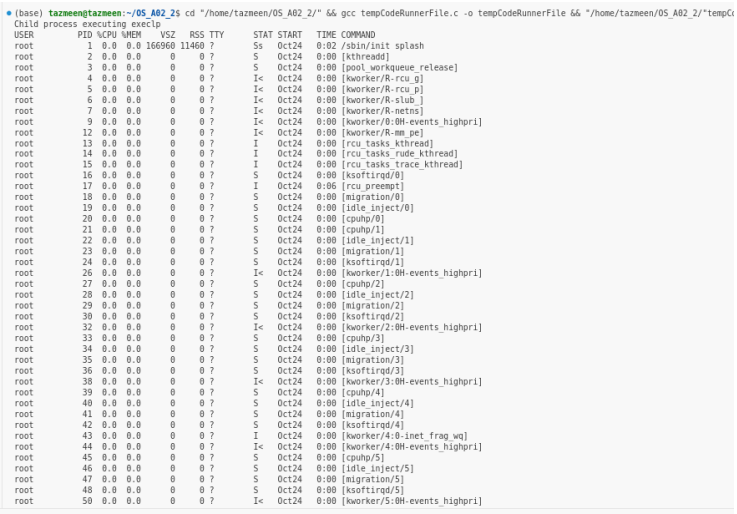

Figure 2: Output of execlp system call.

14 Output of ‘execle‘ System Call

In this case, ‘execle‘ is used to execute the ‘sum‘ program with custom environment variables. The program calculates the sum of integers ”15”, ”25”, and ”35” passed as arguments.

14.1 Code Output

The output of this execution shows the sum of the integers:

14.2 Image of the Output

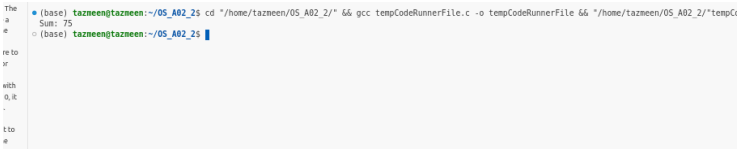

Figure 3: Output of execle system call.

15 Output of ‘execvp‘ System Call

The ‘execvp‘ system call allows the program to execute a command by searching for it in the ‘PATH‘. In this example, we use ‘ps aux‘ to display the list of running processes.

15.1 Code Output

The output shows the running processes similar to the ‘execlp‘ example:

15.2 Image of the Output

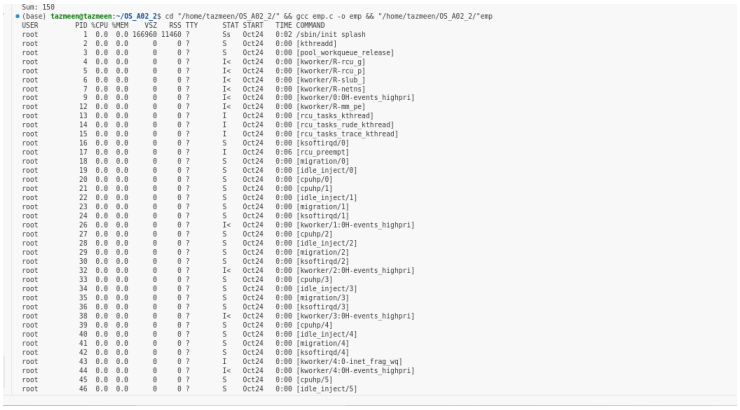

Figure 4: Output of execvp system call.

16 Output of ‘execve‘ System Call

16.1 Image of the Output

Figure 5: Output of execve system call.

17 Output of ‘execv‘ System Call

17.1 Image of the Output

Figure 6: Output of execv system call.

18 Bonus Task: Implementing a Mini-Shell

In this bonus task, we have implemented a simple mini-shell program that accepts user input, parses it into a command and its arguments, and executes the command using the ‘execvp‘ system call.

18.1 Logic of the Program

The mini-shell works by continuously prompting the user for input, parsing the input string into command and arguments, and then executing the command using a child process. Below is a detailed explanation of the steps involved:

- The user is prompted to enter a command along with its arguments.
- The input is read using the ‘fgets‘ function, which takes the user input and stores it in a buffer.
- The ‘parse_input‘ function uses the ‘strtok‘ function to tokenize the input string, breaking it down based on spaces and newline characters to separate the command from its arguments.
- If the command is ‘exit‘, the shell exits by breaking out of the loop.
- Otherwise, the program forks a new child process using the ‘fork‘ system call.
- In the child process, the ‘execvp‘ system call is used to execute the command. This call searches for the executable in the directories listed in the ‘PATH‘ environment variable.
- The parent process waits for the child process to finish execution using the ‘wait‘ function.
- The loop repeats, prompting the user for more commands until they input ‘exit‘.

Key points:

- The ‘strtok‘ function is used to split the input string into command and arguments.
- The ‘fgets‘ function is used to read user input from the terminal.
- ‘execvp‘ is used to execute the parsed command, searching for it in the system's ‘PATH‘.
- Error handling is performed with the ‘perror‘ function in case of failures in ‘fork‘ or ‘execvp‘.

18.2 Code

Below is the implementation of the mini-shell in C:

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/wait.h>
#include <string.h>

void parse_input(char *input, char **args, int arg_count) {
    int i = 0;
    args[i] = strtok(input, " \n");
    while (args[i] != NULL && i < arg_count - 1) {
        i++;
        args[i] = strtok(NULL, " \n");
    }
    args[i] = NULL;
}

void mini_shell(int input_size, int arg_count) {
    char input[input_size];
    char *args[arg_count];
    pid_t pid;

    while (1) {
        printf("mini-shell> ");
        if (fgets(input, input_size, stdin) == NULL) {
            perror("fgets failed");
            exit(1);
        }

        parse_input(input, args, arg_count);

        if (args[0] == NULL) continue;
        if (strcmp(args[0], "exit") == 0) break;

        pid = fork();
        if (pid == 0) {
            execvp(args[0], args);
            perror("execvp failed");
            exit(1);
        } else if (pid > 0) {
            wait(NULL);
        } else {
            perror("fork failed");
        }
    }
}

int main() {
```

```
48      mini_shell(1024, 100);
49      return 0;
50  }
```

18.3 Code Explanation

- The **parse_input** function takes user input and splits it into tokens (command and arguments) using `strtok`.
- The **mini_shell** function handles the main shell loop, which reads input, parses it, and executes it using `fork` and `execvp`.
- The `fgets` function reads user input, ensuring the input size doesn't exceed the buffer limit.
- The `fork` system call creates a new process. The child process runs the command, while the parent waits for it to finish.
- If `execvp` fails, an error message is printed using `perror`, and the child process terminates.

18.4 Output

18.5 Image of the Output

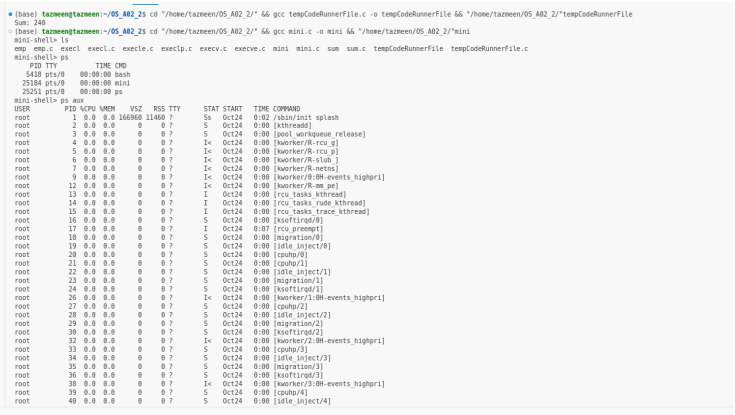

Figure 7: Output of the mini-shell program.