National University of Computer and Emerging Sciences

Laboratory Manual

for

Operating Systems Lab

(CL-2006)

Course Instructor	Ms. Rubab Anam
Lab Instructor	Muhammad Faheem
Section	BCS-4F1
Semester	Spring 2025

Department of Computer Science

FAST-NU, Lahore, Pakistan

Objectives:

- System Calls
- Exec Family
- Pipes

Exec Family of Functions

The exec family of functions replaces the current running process with a new process. It can be used to run a C program by using another C program. It comes under the header file **unistd.h. Many members of the exec family** are shown below with examples. execvp Using this command, the created child process does not have to run the same program as the parent process. The **exec** type system calls allow a process to run any program files, which include a binary executable or a shell script.

Syntax:

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

- **file:** points to the file name associated with the file being executed.
- **argv:** is a null-terminated array of character pointers. Let us see a small example of how to use the execvp() function in C. We will have two .C files, **EXEC.c** and **execDemo.c** and we will replace the execDemo.c with EXEC.c by calling the execvp() function in execDemo.c .

Example

```
//EXEC.c

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

int main()
{
 int i;
 printf("I am EXEC.c called by execvp() ");
printf("\n");

 return 0;
}
```

- Now, create an executable file of EXEC.c using the command

```
gcc EXEC.c -o EXEC
```

```
//execDemo.c
#include<stdio.h>
#include<stdlib.h>
#include<unistd.h> int
main()
{
 //A null terminated array of character
 //pointers
 char *args[]={"./EXEC",NULL};
execvp(args[0],args);

 /*All statements are ignored after execvp() call as this whole
process(execDemo.c) is replaced by another process (EXEC.c) */
 printf("Ending-----");

 return 0;
}
```

- Now, create an executable file of execDemo.c using the command

```
gcc execDemo.c -o execDemo
```

- After running the executable file of execDemo.c by using the command ./excDemo, we get the following output:

```
I AM EXEC.c called by execvp()
```

- When the file execDemo.c is compiled, as soon as the statement execvp(args[0], args) is executed, this very program is replaced by the program EXEC.c. "Ending——" is not printed because as soon as the execvp() function is called, this program is replaced by the program EXEC.c.

execv:

This is very similar to execvp() function in terms of syntax as well. The syntax of **execv()** is as shown below: Syntax

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

- **path:** should point to the path of the file being executed. **argv[]:** is a null-terminated array of character pointers.
  Let us see a small example of how to use the execv() function in C. This example is similar to the example shown above for execvp(). We will have two .C files, **EXEC.c** , and **execDemo.c** and we will replace the execDemo.c with EXEC.c by calling the execv() function in execDemo.c .

Example

```
//EXEC.c

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

int main()
{
 int i;
 printf("I am EXEC.c called by execv() ");
printf("\n"); return 0;
}
```

- Now,create an executable file of EXEC.c using command

gcc EXEC.c -o EXEC

```
//execDemo.c
#include<stdio.h>
#include<stdlib.h>
#include<unistd.h> int
main()
{
 //A null terminated array of character
 //pointers
 char *args[]={"./EXEC",NULL};
execv(args[0],args);

 /*All statements are ignored after execvp() call as this whole
process(execDemo.c) is replaced by another process (EXEC.c) */
printf("Ending-----");


 return 0;
}
```

- Now, create an executable file of execDemo.c using the command

gcc execDemo.c -o execDemo

- After running the executable file of execDemo.c by using the command ./excDemo, we get the following output:

```
I AM EXEC.c called by execv()
```

execlp and execl

These two also serve the same purpose but the syntax them are a bit different which is as shown below: Syntax

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

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

- file: file name associated with the file being executed.
- const char *arg and ellipses: describe a list of one or more pointers to null-terminated strings that represent the argument list available to the executed program. • The same C

programs shown above can be executed with execlp() or execl() functions and they will perform the same task i.e. replacing the current process with a new process.

execvpe and execle:

These two also serve the same purpose but the syntax of them is a bit different from all the above members of the exec family. The syntaxes of both of them are shown below : **Syntax:** int execvpe(const char *file, char *const argv[],char *const envp[]);

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

- The syntaxes above shown have one different argument from all the above exec members, i.e.

  char * const envp[]: allow the caller to specify the environment of the executed program via the argument envp. **envp:** This argument is an array of pointers to null-terminated strings and must be terminated by a null pointer. The other functions take the environment for the new process image from the external variable environ in the calling process.

Reference: https://www.geeksforgeeks.org/exec-family-of-functions-in-c/

Return Value:

The exec() functions return only if an error has occurred. The return value is -1, and errno is set to indicate the error.

Pipes

Ordinary pipes allow two processes to communicate in standard producer-consumer fashion: the producer writes to one end of the pipe (the write-end) and the consumer reads from the other end (the read-end). As a result, ordinary pipes are unidirectional, allowing only one-way communication. If two-way communication is required, two pipes must be used with each pipe sending data in a different direction.

References: Operating System concepts Page no 142 section 3.6.3.1

A pipe has a read end and a write end.
Data written to the write end of a pipe can be read from the read end of the pipe.

Creating a pipe

On UNIX and Linux systems, ordinary pipes are constructed using the function.

- int pipe(int fd[2]) -- creates a pipe.
- returns two file descriptors, fd[0], fd[1].
- fd[0] is the read-end of the pipe.
- fd[1] is the write-end.
- fd[0] is opened for reading,
- fd[1] for writing. pipe() returns 0 on success, -1 on failure, and sets errno accordingly. • The standard programming model is that after the pipe has been set up, two (or more) cooperative processes will be created by a fork and data will be passed using read() and write(). • Pipes opened with pipe() should be closed with
- close(int fd).

Reference: http://linux.die.net/man/2/pipe
Further Reading:
For a general overview of pipes please read **man pipe.7** Read example code in **man pipe.2**

FIFO or Named Pipes

Named pipes or FIFO(First in First out) are persistent pipes. A traditional pipe lasts as long as the process but named pipes can last beyond the life of the process. It is created using mkfifo system call:

Syntax: **int mkfifo(const char *pathname, mode_t mode);**

It makes a special file with name pathname, mode specifies permissions. Do note that it has to open from both ends, reading and writing before data is passed.

Further Reading:

For a general overview of mkfifo in linux https://www.gnu.org/software/coreutils/mkfifo. Also read **man mkfifo.3**

Reference: https://www.geeksforgeeks.org/named-pipe-fifo-example-c-program/

In-Lab Tasks

Question No 1 (Process Trees)

Create a C/C++ program that demonstrates the use of the wait () system call and create process trees. The program should fork exactly 4 child processes from a parent and ensure that the parent process waits for all child processes to complete before proceeding. Also create a child each from the first and last child process. Print process ids of all processes including the parent on screen. Each parent/child i/grandchild i should print **"I am parent/child i/grandchild i process".** The parent process should be the last process to exit.

Question No 2 (Pipes)

Create a C/C++ program that forks a child process. The parent process reads an integer array as a command line argument and sends odd indexed numbers to the child process by a pipe. The child process sends back a parent **sorted array** through another pipe. The parent process sorts the even indexed numbers and merges the sorted array. Both the original array and sorted array should be printed by the parent process at the end. **Child process must be in a separate file and code. You must use the exec system call for using that child process code. File names should be q2parent.c and q2child.c**

Question No 3 (Pipes)

Imagine you are part of a special team working on a cool project! Here's the deal: You've got a teacher, like the leader of the team, and a student, like the helper. The teacher writes a document, like a message for the team, and asks the student to check it for any weird stuff, like odd symbols or numbers. Once the student's done, they give the document back to the teacher for final fixes. It's teamwork at its best! Now, let's write a program that does just that, but with code instead of paper and pen!

Write a C/C++ program named **document_proofreading.c/cpp** that implements the collaborative document editing tool described above. Your program should utilize pipes and **fork()** for communication between the teacher and student processes. The teacher process should be responsible for uploading the document, while the student process should proofread it and send the cleaned version back to the teacher.

Ensure proper error handling for system calls and close unused ends of the pipe in both processes.

Scenario Details:

- The teacher process represents the teacher preparing a document for proofreading.
- The student process represents the student proofreading the document and removing any errors such as **special characters or numbers**. eg #$@ etc
- The communication between the teacher and student processes should allow for effective collaboration and document editing.
- Teacher process must get the document or file through the command line.

Question 4(FIFO Pipes)

Make 2 child processes from a parent process. Each child process must have 2 children so that parent has 4 grandchildren. Parent process should take at most 10 strings, each string a word, from the command line. The first 2 grandchildren must find the minimum and maximum word (based on word length) from the string. The last 2 grandchildren must arrange the strings in alphabetical order. **All grandchildren must send their result to the parent process by using a FIFO pipe**. **Also your code should be like q4parent.c, q4grandchild1.c, q4.grandchild4.c etc. Use the exec system call appropriately to run grandchildren processes.** Eg words from CLI: *apple, banana, rose, tulip, psychology.* Smallest word: *rose* Largest word: *psychology*

Submission Instructions:
Submit your files as q1.c/cpp unless explicitly indicated.