PROJECT :: The xargs Utility in C

Project Objective

Your task is to implement a simplified version of the Unix `xargs` utility in C. This project will help you gain hands-on experience with file input/output, command-line argument parsing, and string manipulation—fundamental skills in systems programming and operating systems development.

It's recommended that you work your way through this tutorial on xargs before you begin programming this project.

Learning Objectives

By completing this project, you will achieve the following learning objectives:

1. **Practice Command-Line Argument Parsing**: Develop skills in parsing and handling command-line arguments in C, including the use of options and flags to modify program behavior.
2. **Handle Input and Output in Unix-like Systems**: Gain proficiency in reading input from standard input, manipulating strings, and building command lines based on input data.
3. **Implement Input Sanitization for Security**: Understand the importance of input sanitization by filtering specific special characters to prevent command injection attacks, reinforcing secure coding practices.
4. **Execute External Commands**: Learn how to use `execvp` to execute external commands based on dynamically generated input, a core aspect of the `xargs` utility.
5. **Reinforce Knowledge of Special Characters in Linux**: Refresh and strengthen your understanding of how special characters like `;`, `&`, `|`, and others are used in Linux, and how they can impact command execution.
6. **Simulate Command-Line Behavior**: Gain experience in simulating the behavior of Unix command-line utilities by building a simplified version of `xargs`, which combines multiple inputs into a single command execution.

These objectives focus on the key aspects of the `xargs` utility, emphasizing input handling, command execution, and security considerations, while providing a solid foundation for further exploration in Unix-like environments.

Background

The `xargs` utility reads items from standard input (or from a file) and executes a command using those items as arguments. This allows you to build and execute command lines from dynamically generated input. The utility is commonly used in combination with other Unix tools like `find` and `grep`.

In this project, you will recreate key features of `xargs`, including:

- Reading input from standard input.
- Splitting input on whitespace and treating each word as a separate argument.
- Building and executing command lines using the input as arguments.
- Handling basic command-line options.

Requirements

1. **Input Handling**: Your program should read input from standard input. Each word of input should be treated as a separate argument unless otherwise specified by the `-I` option.
2. **Command Execution**: The program should construct a command line using the input it reads and execute the command with those arguments. Use `fork` and `execvp` to handle command execution.
3. **Basic Options**:
    - `-n <num>`: Specify the maximum number of arguments to pass to the command at one time.
    - `-I <replace>`: Replace occurrences of `<replace>` in the command with input read from standard input.
    - `-t`: Print the entire constructed command line to standard output before executing it.
    - `-r`: Do not run the command if no input is provided.
4. **Error Handling**: The program should handle errors gracefully, including cases where the command fails or where the input cannot be processed as expected. Ensure that empty input is ignored if the `-r` option is specified.

5. **Security Considerations**:
    - **Sanitize Input**: Your program must filter out specific special characters from the input before it is used in command execution. The characters to filter are:
        - `;` - Used to terminate commands and start new ones.
        - `&` - Used to run commands in the background.
        - `|` - Used to pipe the output of one command to another.
        - `>` - Used to redirect output to a file.
        - `<` - Used to redirect input from a file.
        - `*` - Wildcard character that can match filenames.
        - `?` - Wildcard character that matches a single character.
        - `(` and `)` - Can be used in subshells.
        - `$` - Used to reference shell variables.
    - **Handle Special Characters**: Ensure that your implementation handles special characters and spaces appropriately in input processing.
    - **Avoid Direct Shell Execution**: Ensure that the command execution is secure and only the intended command and arguments are executed, without invoking unintended shell operations.

Introduction to `fork` and `execvp`

In this section, we will explore two key system calls, `fork` and `execvp`, that you'll use in your `xargs` project. These system calls are fundamental to Unix-like operating systems and are widely used to create and manage processes. Understanding them will provide you with the necessary tools to implement the core functionality of your project.

`fork`: Creating a New Process

The `fork` system call is used to create a new process. When you call `fork`, the operating system creates a duplicate of the current process. This duplicate is known as the child process, while the original is called the parent process.

Key Points:

- **Return Values**: `fork` returns a value that allows the program to determine whether it's in the parent or the child process.
  - **Parent Process**: `fork` returns the Process ID (PID) of the child process.
  - **Child Process**: `fork` returns `0`.
  - **Failure**: `fork` returns `-1` if the process creation fails.
- **Execution Flow**: After a successful `fork`, both the parent and child processes continue executing from the point where `fork` was called, but they have separate memory spaces.

Example Usage:

```
pid_t pid = fork();

if (pid < 0) {
    // Fork failed
    perror("fork");
    exit(1);
} else if (pid == 0) {
    // Child process
    printf("This is the child process.\n");
} else {
    // Parent process
    printf("This is the parent process. Child PID: %d\n", pid);
}
```

`execvp`: Executing a Command

The `execvp` system call is used to replace the current process image with a new process image. It loads and runs a new program, specified by the given filename, and does not return unless an error occurs. The `execvp` call is part of the `exec` family of functions, where the `vp` suffix stands for "vector of arguments" and "search PATH."

Key Points:

- **Command and Arguments:** `execvp` takes two arguments:
  - **Filename**: The name of the file to be executed.
  - **Argument Vector**: An array of strings (arguments), where the first element is the name of the command, and the last element is `NULL`.
- **PATH Search:** `execvp` searches for the file in the directories listed in the `PATH` environment variable.
- **No Return**: If `execvp` is successful, it does not return. The current process is replaced by the new process. If it fails, it returns `-1`.

Example Usage:

```
char *args[] = {"ls", "-l", "/home", NULL};
execvp(args[0], args);

// If execvp returns, it must have failed
perror("execvp");
exit(1);
```

Combining `fork` and `execvp`

In many applications, you'll use `fork` and `execvp` together. The typical pattern is to `fork` a child process and then use `execvp` in the child process to run a new program.

Example Usage:

```
pid_t pid = fork();

if (pid < 0) {
    // Fork failed
    perror("fork");
    exit(1);
} else if (pid == 0) {
    // Child process: replace with new program
    char *args[] = {"ls", "-l", "/home", NULL};
    execvp(args[0], args);
    // If execvp returns, it must have failed
    perror("execvp");
    exit(1);
} else {
    // Parent process: wait for the child to finish
    wait(NULL);
    printf("Child process finished.\n");
}
```

Summary

- `fork` is used to create a new process (child process). Both parent and child processes continue execution from the point of `fork`, with `fork` returning different values to each.
- `execvp` is used to replace the current process image with a new program. It takes the command to execute and its arguments, and it doesn't return if successful.
- **Combining `fork` and `execvp`** allows you to create a new process to run a different program, which is exactly what you need to do for your `xargs` project.

This foundational understanding will enable you to implement the process creation and command execution features required for the project. As you work through the project, remember that these tools will be revisited in more detail later in the course when we dive into process management. For now, this should give you a solid starting point.

C Language Compilation and Execution Instructions

Compilation

Your code must be compiled using `gcc`, the GNU Compiler Collection. Ensure that your code compiles on a recent version of Ubuntu. Here's how you should compile your `xargs` program:

```
gcc -o myxargs myxargs.c
```

This command compiles your `myxargs.c` file and produces an executable named `myxargs`.

When compiling your C program, it's highly recommended to use the `-Wall` and `-Werror` options with `gcc`. The `-Wall` flag enables all the important compiler warnings, helping you identify potential issues in your code, such as unused variables, questionable syntax, or implicit type conversions. The `-Werror` flag treats all warnings as errors, forcing you to address them before the code can compile successfully. Using these options will help you write more robust and error-free code by catching potential problems early in the development process. Here's how you can include these flags in your compilation command:

```
gcc -Wall -Werror -o myxargs myxargs.c
```

This command ensures that your code is as clean and reliable as possible, which is especially important for systems programming projects like this one.

Execution

After compiling your program, you should test it on a late version of Ubuntu. The following is an example of how to run your `xargs` implementation and what the expected output should look like.

Usage Example

If the program is run without sufficient arguments or with an incorrect configuration, it should display a usage message. Here's how that should look:

```
./myxargs
```

Expected Output:

```
Usage: myxargs [-n num] [-I replace] [-t] [-r] command
```

This message should be displayed anytime the user fails to provide the correct options or arguments. It helps users understand the required and optional parameters for running your program.

Example Run

Here's an example of how your program might be run, along with the expected behavior:

1. **Basic Execution Example**:

```
echo "file1.txt file2.txt file3.txt" | ./myxargs -t rm
```

Expected Behavior: Your program should output the constructed command line (`rm file1.txt file2.txt file3.txt`) before executing it. The files `file1.txt`, `file2.txt`, and `file3.txt` would then be removed.

2. **Using the `-n` Option**:

```
echo "file1.txt file2.txt file3.txt" | ./myxargs -n 1 -t rm
```

Expected Behavior: Your program should print and execute the `rm` command for each file individually:

```
+ rm file1.txt
+ rm file2.txt
+ rm file3.txt
```

3. **Using the `-I` Option**:

```
echo "apple orange banana" | ./myxargs -I {} -t echo Fruit: {}
```

Expected Behavior: Your program should print and execute the `echo` command with all fruit substituted into the placeholder:

```
```

```
+ echo Fruit: apple orange banana
Fruit: apple orange banana
```

4. **Using the `-r` Option**:

```
echo "" | ./myxargs -r -t rm
```

Expected Behavior: If the input is empty, the program should not execute the `rm` command, and nothing should be printed.

Compare this to the following:

```
echo "" | ./myxargs -t rm
```

Expected Behavior: If the input is empty, but -r is not included, then the program should execute the `rm` command.

```
+ rm
rm: missing operand
Try 'rm --help' for more information.
```

Make sure your implementation matches these expectations to ensure consistent and correct behavior. This guidance will help ensure your code compiles and runs as expected in a standardized environment.

Grading Criteria

Your project will be evaluated based on the following criteria:

1. **Correctness and Testing (70%)**:
    - Does the program behave as expected when tested against grading test cases?
    - Does it correctly implement all required features and options (`-n`, `-I`, `-t`, `-r`)?
    - Is the program robust against various input scenarios, including edge cases?
    - Does the program handle input and output appropriately, and does it correctly sanitize and execute commands?
2. **Code Quality and Functional Decomposition (20%)**:
    - Is the code well-structured and modular, with clear functional decomposition?
    - Are functions properly broken down into smaller, manageable tasks rather than being overly complex?
    - Are variables and functions named appropriately, following standard coding conventions?
    - Is the code adequately documented with comments that explain the purpose of each section?
3. **Compilation and Execution (10%)**:
    - Does the program compile without **errors** or **warnings** using the specified compilation command?
    - Does the program run as expected in the target environment (Ubuntu)?
    - If the program fails to compile or execute correctly, it cannot receive more than 10% of the grade, which will be based purely on code quality. No other test cases will be considered for code that fails to compile or execute.

Important Note: You will have access to a GradeScope based checker tool that will help you verify whether your program compiles and runs. This tool will not run all grading test cases, it's primarily intended to make sure that you are not omitting header files. Make sure to test your code with this tool before submission to ensure that it meets the basic requirements.

This grading rubric emphasizes the importance of correctness, functional decomposition, and clean, modular code. Programs that do not compile or run successfully can only earn points for code quality, with a maximum of 10% of the total grade.

Submission Details

You will be submitting a single C file named `myxargs.c` for this project. Most likely you will submit with GradeScope. Details will be posted next week.

Release History

Just a friendly reminder: Keeping up with the latest version of assignments is key! We might update them now and then to iron out any bugs or to make things clearer. Whenever there's an update, we'll let you know through a Canvas announcement, so please make sure you're set up to receive those notifications. Embrace the dynamic nature of programming, where adapting to evolving requirements is part of the adventure. If you spot any bugs in the assignments, feel free to send me a bug report. It's a great way to help out and contribute to our course community!

V002 : Edited some incorrect behaviors and add some more detail.
V001 : Initial release, expect a few bugs.