5DV088 - Systemnära Programmering

PipeSim: Simulating Pipelines of Unix Pipes

version 1.0

Name	Rasmus Mikaelsson
User	et24rmn

Personnel
Jonny Pettersson, jonny@cs.umu.se
Algot Heimerson, dv23ahn@cs.umu.se
Willy Själin, willys@cs.umu.se
Linus Svedberg, oi22lsg@cs.umu.se
Isak Öhman, c21ion@cs.umu.se

Contents

1 Name

Full name: Rasmus Mikaelsson
User: et24rmn

2 Synopsis

```
mexec [file]
```

The program can be run interactively (without arguments) or by providing a text file containing commands.

3 Description

`mexec` is a program that enables communication between processes using the **pipe()** system call. It simulates the functionality of a Unix terminal pipeline by creating child processes and connecting them so that the output from one process is passed as input to the next. For example, the command:

```
cat -n mexec.c | grep -B4 -A2 return | less
```

demonstrates multiple programs executed in sequence, where each command's output is forwarded to the next using the pipe symbol (|). `mexec` replicates this behavior by handling the creation and linking of child processes through **pipe()** system calls.

4 Solution

`mexec` reads commands from `stdin` or from a text file given as the first argument. Each command must be on a separate line and blank lines are not allowed. There is no upper limit on the number of commands, and each line can be at most 1024 characters long. The program parses the commands, creates child processes, and executes them in sequence.

Figure 1 illustrates how the child processes are connected through pipes, showing the flow of data from the first command to the last.

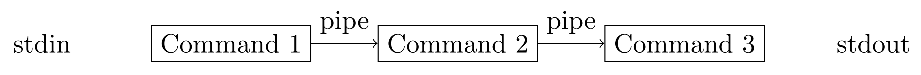

Figure 1: Child processes connected through pipes in `mexec`. Each command's output is sent to the next command's input.

4.1 Main Algorithm / Flow of the program

The program executes commands in a pipeline according to the following steps:

1. **Input Handling:**
   - If a filename is provided as the first argument, open the file and read commands line by line.
   - If no file is provided, read commands from `stdin`.
   - Validate that the file exists and contains readable commands. Exit on error.
2. **Command Parsing:**
   - Read all commands into memory and count them.
   - Each line is parsed into a command and its arguments.
3. **Pipeline Setup and Process Creation:**
   - For each command in the sequence:
     (a) Create a pipe to connect the current command's output to the next command's input (if it is not the last command).
     (b) Fork a child process.
     (c) In the child:
       – Redirect input from the previous command's pipe (or `stdin` for the first command).
       – If not the last command, redirect output to the current pipe.
       – Execute the command using `execvp()`.
       – Exit with an error if execution fails.
     (d) In the parent:
       – Close unused pipe ends to avoid resource leaks.
       – Wait for the child to finish before proceeding to the next command.
4. **Cleanup:**
   - Close any remaining open file descriptors.
   - Free all allocated memory used to store commands.
   - Exit with status 0 if all commands executed successfully, or with an appropriate error code if any step failed.

4.2 Error Handling and Cleanup

The program includes comprehensive error handling to ensure safe execution and resource management. Common errors handled includes:

- **File Errors:** If a specified input file cannot be opened, the program prints an error message and exits.
- **Memory Allocation Errors:** If memory allocation or reallocation fails while storing commands, the program prints an error message, frees any previously allocated memory, and exits.

- **System Call Failures:** If any `fork()`, `pipe()`, `dup2()`, or `execvp()` call fails, the program prints an appropriate error message and exits.

All child processes are waited for by the parent to ensure proper termination. Any open pipe file descriptors are closed when no longer needed, and dynamically allocated memory is freed before the program exits. This prevents resource leaks and ensures that the program terminates cleanly even in the case of errors.

4.3 Compilation and Execution

The program is written in C and includes a `Makefile` for convenient compilation. To obtain and run the project, follow these steps:

1. Clone the project repository:

   ```
   git clone https://github.com/rasmusmikaelsson/mexec.git
   ```

2. Compile the program:

   ```
   make
   ```

3. Run the program without a file argument (interactive mode):

   ```
   ./mexec
   ```

4. Run the program with a text file containing commands:

   ```
   ./mexec [file]
   ```

5 Exit Status

The program returns the following exit codes:

- **0** – Successful execution; all commands executed correctly.
- **1 (EXIT_FAILURE)** – An error occurred. Possible causes include:
  - Too many command-line arguments.
  - Input file could not be opened.
  - Memory allocation or reallocation failure.
  - `fork()`, `pipe()`, `dup2()`, or `execvp()` failure.
  - Failure to parse a command line.

6 Discussion and reflection

During the development of `mexec`, the main challenges were understanding how `fork()` and `pipe()` work. In particular, grasping the process of creating child processes and the benefits and limitations of using `fork()` required careful study. Connecting child processes with pipes to form a complete pipeline was by far the most difficult part of implementing `mexec`.

A reflection I have noticed is that this time around, I started the assignment early, which gave me sufficient time to understand its key concepts. While I would not call myself an expert, I now have a solid grasp of how file descriptors work, the basics of creating child processes using `fork()`, and how connecting processes with pipes enables inter-process communication. This understanding is the main takeaway from this assignment and I am sure that I will be revisiting these concepts throughout the remainder of this course, as well in the future.