Programming Project 1

Instructions for this assignment:

You may work on this assignment either solo, or in a group of up to four students (including yourself). Submit the assignment by following the steps listed under "Submission Procedure" before the due date/time; late submissions will receive a grade of zero, but on-time submissions will receive partial credit even if incomplete. If working in a group, each member of the group should submit the assignment, and enter the names of all group members in the Comments field in Blackboard.

This project is intended to introduce you to the elements of concurrent programming using the POSIX standard API. Most Unix and Linux systems adhere closely to this API, and these or similar systems are heavily used in backend applications and cloud computing. If you're already familiar with Linux, some of the instructions will seem rather simplistic!

For this project, you may use an Ubuntu Linux Virtual Machine (provided by the authors of our required textbook) running on the VirtualBox application, or an Ubuntu or other Linux-compatible system such as the CS department servers (cs.uml.edu).

Problem Statement

In this part of the project, you are going to redirect Linux input within a C program after forking a child process and using the `execlp` system call. I'll first provide some background.

The Linux/UNIX shell command line environment provides a large number of simple commands that process text files. Suppose that you have a text file, `Names.txt`, that contains a series of names (one per line of the file), and that you want to know how many names are in the file. You can type:

```
    wc -l < Names.txt
```

which will display the number of lines in `Names.txt`, which is the number of names.

However, if you examine `Names.txt`, you'll see that it contains a lot of duplicate names. If you want to know the number of unique names in `Names.txt`, you can use the Linux shell's pipe redirector to use several commands in a pipeline, as below:

```
    sort < Names.txt | uniq | wc -l
```

The `sort` command is given the list of names and sorts them. The `uniq` command removes duplicate lines in a file, but only if the duplicate lines are adjacent. By sorting `Names.txt` and piping the sorted list to `uniq` (using the "`|`" symbol), the duplicate lines are adjacent and `uniq` can remove them. Finally, the sorted list with duplicates removed is piped to the `wc` command, which counts the number of lines of the sorted list with the duplicates removed to display the number of unique names.

You will write a small C program which does what the Linux shell does, without doing the command line processing. That is, you'll write a program that creates three child processes, one for each command. Each child process will tie an end of a pipe to the standard input, standard output, or both, and then call the `execlp` system call to run the command. The three child processes will be connected by two Linux pipes, one pipe between the first two processes and the other pipe between the second two processes. The overall structure of the program is shown in Figure 1. Figures 2 and 3 show examples discussed in class which show the use of `execlp` to run a Linux command and the creation of pipes. Your program will use the techniques from both of these programs.

However, the code in Figure 3, which demonstrates pipes, doesn't handle the redirection needed. The problem is that once the `execlp` finishes, each child process is running a brand-new program image which knows nothing of the pipes that were created by the parent process. So, before calling `execlp` to start the new program, each child must first tie the appropriate ends of the pipes to its standard input and/or output using the `dup2` system call.

Your program will need to use the following include statements:

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

You will also need the following POSIX API/Linux system calls and C library calls:

-
  ```
  printf(<format-string>, arg0, arg1, ..., argn);
  fprintf(<file-pointer>, <format-string>, arg0, arg1, ..., argn);
  ```

  `printf` and `fprintf` allow you to display formatted output. The only difference between them is that `printf` always displays its output on the standard output (`stdout`), while `fprintf` allows you to choose the output file (for this assignment, the only need for `fprintf` is to send output to the standard error output, `stderr`). Replace `<file-pointer>` with the file pointer to which you want to send the output. The two standard output file pointers are `stdout` and `stderr`. The `<format-string>` is a C-language literal string that contains the text to send to output with embedded placeholders (`%d`, integer value, `%s`, string value, `%f`, floating point value, etc.). Each placeholder in the format string is replaced with one of the remaining arguments (`arg0`, `arg1`, etc.) in the call. There should be as many arguments as there are placeholders in the format string.

The call:

```
printf("This process id is %d\n", getpid());
```

displays the string:

```
This process id is 19334
```

on a separate line, if the function `getpid()` returns 19334 as the process ID for the current process.

- `getpid()`
  This system call returns the process ID (pid) of the current process.

- `fork()`
  This system call creates a new process that is an exact duplicate of the current process. The only difference between the two processes are the return values from `fork()`: The child process sees a return value of zero, while the parent process sees the PID of the child process.

- `close(<fd>)`
  This system call closes the file descriptor `<fd>` for output. You do not need to check for errors.

- `pipe(<fd-array>)`
  This system call places two file descriptors into the two-element integer array `<fd-array>` corresponding to the ends of an open pipe. Element zero of `<fd-array>` is the read end of the pipe, and element one is the write end of the pipe.

- `dup2(<oldfd>, <newfd>)`
  This system call ties the existing file descriptor `<oldfd>` to the file descriptor `<newfd>`. After the call, `<oldfd>` and `<newfd>` refer to the same file descriptor. If `<newfd>` is already open, the current output is closed and then `<newfd>` is tied to `<oldfd>`. For this part of the project, `<oldfd>` will be either the read or write end of one of the pipes (`fd1` or `fd2`), and `<newfd>` will be either 0 (standard input) or 1 (standard output). You do not need to check for errors.

-
  ```
  execlp(<command-path-name>, <command-name>, <arg-string-1>,
         <arg-string-2>, ..., NULL);
  ```
  This system call replaces the current program image (code, data, and stack) with the image found at the path name `<command-path-name>`. After this call is complete, the process is running a completely different program with new code, data, and stack. `<command-name>` is the simple name for the command, which is placed in the command line argument zero. `<arg-string-1>, <arg-string-2>`, etc. are command line arguments 1, 2, etc. The command line argument strings are C-language pointers to character arrays (as are C-language

string literals). The set of command line arguments must end with a zero or `NULL` pointer. If there are no arguments for the command, simply use `NULL`. The parameters to `execlp` for each of the commands are as follows:

- `sort`
  - `<command-path-name>` is `"/usr/bin/sort"`
  - `<command-name>` is `"sort"`
  - no `<arg-string-x>`, only `NULL`
- `uniq`
  - `<command-path-name>` is `"/usr/bin/uniq"`
  - `<command-name>` is `"uniq"`
  - no `<arg-string-x>`, only `NULL`
- `wc`
  - `<command-path-name>` is `"/usr/bin/wc"`
  - `<command-name>` is `"wc"`
  - `<arg-string-1>` is `"-l"`
  - `NULL`

- `waitpid(<pid>, NULL, 0);`
  This system call waits for the child process with the process id `<pid>`. The `NULL` second argument indicates that no status information is to be returned. The third argument is an integer representing various options (see the `man` page for `waitpid` for possible values) – 0 indicates no options are selected. (Note: The `waitpid` system call is a better choice for this project than its simpler cousin `wait` !)

We're now ready to begin work on this step of the project.

1. To start this step of the project, open a terminal window and type:

   ```
   mkdir SortUniqWc
   ```

   and then:

   ```
   cd SortUniqWc
   ```

2. You will now be in the directory `SortUniqWc`. Type:

   ```
   gedit sortuniqwc.c &
   ```

   to open the `gedit` editor (**or open your favorite editor**).[1] Use the output provided in Figure 1 and the examples shown in Figures 2 and 3 to guide you.

   It's a good idea to add `printf` statements to each of the children to display their process IDs:

   ```
   printf("The child process running <cmd> is %d\n",
          getpid());
   ```

   where `<cmd>` is the name of the command this child is running.

   Also add a `printf` after each `execlp`:

   ```
   printf("Should not be here after execlp to <cmd>\n");
   ```

   If `execlp` fails, the `printf` will display a message; if the `execlp` succeeds, the `printf` will not execute. This is a simple way to verify that `execlp` is working. Typical reasons for `execlp` failing are a bad command path name, or a bad argument sequence (most commonly, a missing `NULL` at the end of the command line arguments).

   Also, add a `printf` to the main process code that informs you when the last child is finished (just after the call to `wait`).

3. From the Blackboard page for this assignment, download the file, `Names.txt`, that contains the list of names.[2]

[1] Note that `gedit` requires a GUI. If you're using a terminal/command line only system, then a screen-based text editor such as `vi` or `emacs` can be used instead.
[2] If you are using an Ubuntu VM, you can import this file either by using the shared folder facility, or by setting the virtual machine to accept copy and paste from the host OS. For the latter, click on VirtualBox's `Devices` menu and select `Drag and Drop->Bidirectional`.

4. Type the command:

   ```
   sort < Names.txt | uniq | wc -l
   ```

   in your terminal. You should get a display of 23, indicating that there are 23 unique names in the `Names.txt` file. (If you get a value other than 23, use an editor to check whether there is a superfluous line at the end of your copy of the `Names.txt` file.)

5. To run your program, you must first compile it. To do so, type

   ```
   gcc -o sortuniqwc sortuniqwc.c
   ```

   This command will compile the program in `sortuniqwc.c` and put the binary executable in the file `sortuniqwc`.

6. To run your program, type:

   ```
   ./sortuniqwc < Names.txt
   ```

   The "`./`" tells the shell to execute the program in the file `sortuniqwc` by looking in the current directory (`SortUniqWc`) rather than searching through the standard system paths. You should see the printed messages you added to the code. There is no certain order to these messages; running the program several times may result in slightly different orders. This is normal.

Submission Procedure

Submit your source file `sortuniqwc.c` using the Blackboard page for this assignment.

Remember to indicate in the Comments field for the submission whether you worked solo or in a group, and (if you worked in a group) list all of your group members including yourself.

Outline for your program

```
/*   insert #include directives for all needed header files here  */


int main(int argc, char *arv[]) {

       //create first pipe fd1
       // fork first child
       pid = fork();  // create first child for sort
       if (pid < 0) {
              // fork error
       }
       if (pid == 0) { // first child process, run sort
              // tie write end of pipe fd1 to standard output (file descriptor 1)
              // close read end of pipe fd1
              // start the sort command using execlp
              // should not get here
       }
       //create second pipe fd2
       // fork second child
       pid = fork(); // create second child for uniq
       if (pid < 0) {
              // fork error
       }
       if (pid == 0) { // second child process, run uniq
              // tie read end of fd1 to standard input (file descriptor 0)
              // tie write end of fd2 to standard output (file descriptor 1)
              // close write end of pipe fd1
              // close read end of pipe fd2
              // start the uniq command using execlp
              // should not get here
       }
       // fork third child
       pid = fork() // create third child for wc -l
       if (pid < 0) {
              // fork error
       }
       if (pid == 0) { // third child process, run wc -l
              // tie read end of fd2 to standard input (file descriptor 0)
              // close write end of pipe fd2
              // close read end of pipe fd1
              // close write end of pipe fd1
              // start the wc -l command using execlp
              // should not get here
       }
       // parent process code
              // close both ends of pipes fd1 and fd2
              // wait for third process to end.
}
```

Figure 1: Outline of the `sortuniqwc.c` *program*

Example of using **`execlp`**

```
/**
 * This program forks a separate process using the fork()/exec() system calls.
 *
 * Figure 3.09
 *
 * @author Silberschatz, Galvin, and Gagne
 * Operating System Concepts  - Ninth Edition
 * Copyright John Wiley & Sons - 2013
 */

#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>

int main()
{
pid_t pid;

	/* fork a child process */
	pid = fork();

	if (pid < 0) { /* error occurred */
		fprintf(stderr, "Fork Failed\n");
		return 1;
	}
	else if (pid == 0) { /* child process */
		printf("I am the child %d\n",pid);
		execlp("/bin/ls","ls",NULL);
	}
	else { /* parent process */
		/* parent will wait for the child to complete */
		printf("I am the parent %d\n",pid);
		wait(NULL);

		printf("Child Complete\n");
	}

    return 0;
}
```

Figure 2: Figure 3.9 from OSC textbook showing use of `execlp` to run the Linux command

Example using the pipe system call

```
/**
 * Example program demonstrating UNIX pipes.
 *
 * Figures 3.25 & 3.26
 *
 * @author Silberschatz, Galvin, and Gagne
 * Operating System Concepts  - Ninth Edition
 * Copyright John Wiley & Sons - 2013
 */

#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>
#include <string.h>

#define BUFFER_SIZE 25
#define READ_END 0
#define WRITE_END        1

int main(void)
{
        char write_msg[BUFFER_SIZE] = "Greetings";
        char read_msg[BUFFER_SIZE];
        pid_t pid;
        int fd[2];

        /* create the pipe */
        if (pipe(fd) == -1) {
                fprintf(stderr,"Pipe failed");
                return 1;
        }

        /* now fork a child process */
        pid = fork();

        if (pid < 0) {
                fprintf(stderr, "Fork failed");
                return 1;
        }

        if (pid > 0) {  /* parent process */
                /* close the unused end of the pipe */
                close(fd[READ_END]);

                /* write to the pipe */
                write(fd[WRITE_END], write_msg, strlen(write_msg)+1);

                /* close the write end of the pipe */
                close(fd[WRITE_END]);
        }
        else { /* child process */
                /* close the unused end of the pipe */
                close(fd[WRITE_END]);

                /* read from the pipe */
                read(fd[READ_END], read_msg, BUFFER_SIZE);
                printf("child read %s\n",read_msg);

                /* close the write end of the pipe */
                close(fd[READ_END]);
        }

        return 0;
}
```

Figure 3: Figures 3.25 and 3.26 from OSC textbook showing use of Linux pipes