CS 332/532 – 1G- Systems Programming

Lab 10

Objectives

- Review Linux I/O Streams
- Discuss sharing between parent and child processes
- Review copying file descriptors – dup2() system call

Linux I/O Streams

Before we discuss I/O redirection, let us review how I/O is handled in Linux systems. Each process in a Linux environment has three different file descriptors available when a process is created: standard input (*stdin – 0*), standard output (*stdout – 1*), and standard error (*stderr – 2*). These three file descriptors are created when a process is created. We use the *stdin* file descriptor to read input from a keyboard or from another a file or from another program. Similarly, we use the *stdout* and *stderr* file descriptors to write output and error messages, respectively, to the terminal.

You have already been using these file descriptors when you wrote the insertion sort program in C. You read the number of elements and the elements to be sorted from the keyboard using the *scanf* function. The *scanf* function was using *stdin* file descriptor to read your keyboard input. In other words, the following two functions are equivalent:

```
scanf("%d", &N);
fscanf(stdin, "%d", &N);
```

Similarly when you use the *printf* function to print the output of your program you are using the *stdout* file descriptor. The two functions below are equivalent:

```
printf("%d\n", N);
fprintf(stdout,"%d\n", N);
```

The file descriptors *stdin*, *stdout*, and *stderr* are defined in the header file *stdio.h*. We typically use the *stderr* file descriptor to write error messages.

If we need to save the output or error message from a program to a file or read data from a file instead of entering it through the keyboard, we can use the I/O redirection supported by the Linux shell (such as bash). In fact, we already used this in one of the earlier labs when we used insertion sort to sort large input values. The following examples show how to use I/O redirection in the bash shell to read input from a file (instead of entering it from the keyboard) and send the output and error messages to

different files (instead of the terminal). You can download the program `myprog.c` create a file called *input.txt*, type you name in the file *input.txt*, compile the program (use: *gcc -Wall -o myprog myprog.c*), and test it as shown below (lines starting with # are just comments, you don't have to type these lines).

```
#read input from file input.txt, write output and error messages to terminal
$ ./myprog < input.txt

#read input from file input.txt, write output to file output.txt and error me
ssages to terminal
$ ./myprog < input.txt > output.txt

#read input from file input.txt, write output to terminal and error messages
to file error.txt
$ ./myprog < input.txt 2> error.txt

#read input from file input.txt, write output to file output.txt and error me
ssages to file error.txt
$ ./myprog < input.txt > output.txt 2> error.txt

#read input from file input.txt, write both output and error messages to file
output.txt
$ ./myprog < input.txt &> output.txt
```

You can replace **>** with **>>** in the above examples if you like to append to the file instead of overwriting the file.

Sharing between parent and child processes

When we created a new process using fork/exec in the previous labs, we noted that the child process is a copy of the parent process and it inherits several attributes from the parent process such as open file descriptors. This duplication of descriptors allowed the child processes to read and write to the standard I/O streams (note that the child process was able to read input from the keyboard and write output to the terminal). As a result of this sharing both parent and child processes share the three standard I/O streams: *stdin*, *stdout*, and *stderr*.

Let us use the example from the previous lab to illustrate this by adding the following lines in the parent process:

```
        char name[BUFSIZ];

        printf("Please enter your name: ");
        scanf("%s", name);
        printf("[stdout]: Hello %s!\n", name);
        fprintf(stderr, "[stderr]: Hello %s!\n", name);
```

You can download the complete program here: `forkexecvp2.c`

If we compile and execute the program by using *myprog* (used earlier) as the child process (for example: *./a.out ./myprog*) you will notice that the prompt to enter the name is printed twice. If you enter the name, which process is reading the keyboard input? You could add the PID in the printf statement to make this explicit. You can update the code above to print the PID and test the program. In any case, this illustrates the result of sharing of the standard I/O streams between the parent and the child processes.

If we like to change the behavior of all child processes to use separate files instead of the standard I/O streams we have to replace the standard I/O file descriptors with new file descriptors. We will use the dup2() system call to create a copy of this file descriptors and associate separate files to replace the standard I/O streams.

Copying file descriptors – dup2() system call

The dup2() system call duplicates an existing file descriptor and returns the duplicate file descriptor. After the call returns successfully, both file descriptors can be used interchangeably. If there is an error then -1 is returned and the corresponding errno is set (look at the man page for dup2() for more details on the specific error codes returned). The prototype for the dup2() system call is shown below:

```
#include <unistd.h>

int dup2(int oldfd, int newfd);
```

The following example illustrate how to use dup2 to replace the stdin and stdout file descriptors with the files stdin.txt and stdout.txt, respectively. We use the file `forkexecvp.c` from the previous lab and the new lines of code are highlighted. You can download the complete file here: `ioredirect.c`

```
/* Simple program to illustrate the implementation of I/O redirection.
 * This version uses execvp and command-line arguments to create a new proces
s.
 * To Compile: gcc -Wall -o ioredirect ioredirect.c
 * To Run: ./ioredirect <command> [args]
 * The new child process created executes the program <command> by reading
 * input from the file stdin.txt and writes output to the file stdout.txt.
 */

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

```c
#include <fcntl.h>

int main(int argc, char **argv) {
    pid_t pid;
    int status;
    int fdin, fdout;

    /* display program usage if arguments are missing */
    if (argc < 2) {
        printf("Usage: %s <command> [args]\n", argv[0]);
        exit(-1);
    }

    /* open file to read standard input stream,
       make sure the file stdin.txt exists, even if it is empty */
    if ((fdin = open("stdin.txt", O_RDONLY)) == -1) {
       printf("Error opening file stdin.txt for input\n");
       exit(-1);
    }

    /* open file to write standard output stream in append mode.
       create a new file if the file does not exist. */
    if ((fdout = open("stdout.txt", O_CREAT | O_APPEND | O_WRONLY, 0755)) ==
-1) {
       printf("Error opening file stdout.txt for output\n");
       exit(-1);
    }

    pid = fork();
    if (pid == 0) { /* this is child process */
        /* replace standard input stream with the file stdin.txt */
        dup2(fdin, 0);

        /* replace standard output stream with the file stdout.txt */
        dup2(fdout, 1);

        execvp(argv[1], &argv[1]);
        /* since stdout is written to stdout.txt and not the terminal,
           we should write to stderr in case exec fails, we use perror
           that writes the error message to stderr */
        perror("exec");
        exit(-1);
    } else if (pid > 0) { /* this is the parent process */

        /* output from the parent process still goes to stdout :-) */
        printf("Wait for the child process to terminate\n");

        wait(&status); /* wait for the child process to terminate */
        if (WIFEXITED(status)) { /* child process terminated normally */
```

```
            printf("Child process exited with status = %d\n", WEXITSTATUS(sta
tus));
            /* parent process still has the file handle to stdout.txt,
               now that the child process is done, let us write to
               the file stdout.txt using the write system call */

            write(fdout, "Hey! This is the parent process\n", 32);
            close(fdout);

            /* since we opened the file in append mode, the above text
               will be added after the output from the child process */
        } else { /* child process did not terminate normally */

            printf("Child process did not terminate normally!\n");
            /* look at the man page for wait (man 2 wait) to determine
               how the child process was terminated */
        }
    } else { /* we have an error */

        perror("fork"); /* use perror to print the system error message */
        exit(EXIT_FAILURE);
    }

    return 0;
}
```

Compile and run this program using myprog as the child process. Note that you have provide input to *myprog* in the file *stdin.txt* and the output of *myprog* will be written to *stdout.txt*. As we did not do anything with the *stderr* stream, the output to *stderr* stream goes to the terminal. You can add the PID in the print statement to confirm which output is sent to the terminal and which output is sent to the files. Here is a terminal session that illustrates this interaction:

```
$ gcc -Wall -o myprog myprog.c
$ gcc -Wall -o ioredirect ioredirect.c
$ cat > stdin.txt
Bangalore
$ ./ioredirect ./myprog
Wait for the child process to terminate
[stderr]: Hello Bangalore!
Child process exited with status = 0
$ cat stdout.txt
Please enter your name: [stdout]: Hello Bangalore!
Hey! This is the parent process
$
$ cat > stdin.txt
World
```

```
$ ./ioredirect ./myprog
Wait for the child process to terminate
[stderr]: Hello World!
Child process exited with status = 0
$ cat stdout.txt
Please enter your name: [stdout]: Hello Bangalore!
Hey! This is the parent process
Please enter your name: [stdout]: Hello World!
Hey! This is the parent process
$
$ ./ioredirect uname -a
Wait for the child process to terminate
Child process exited with status = 0
$ cat stdout.txt
Please enter your name: [stdout]: Hello Bangalore!
Hey! This is the parent process
Please enter your name: [stdout]: Hello World!
Hey! This is the parent process
Linux vulcan16.cis.uab.edu 3.10.0-862.3.3.el7.x86_64 #1 SMP Fri Jun 15 04:15:
27 UTC 2018 x86_64 x86_64 x86_64 GNU/Linux
Hey! This is the parent process
$
```

Homework

This homework is very similar to Lab-7 homework assignment, the only change required is highlighted below. You can start with the Lab-7 homework solution: `lab7_solution.c`

Write a program that takes a filename as a command-line argument and performs the following steps:

1. Open the file specified as the command-line argument.
2. Read contents of the file one-line at a time.
3. Use fork-exec to create a new process that executes the program specified in the input file along with the arguments provided.
4. The child process will redirect the standard output stream (stdout) to a file *<pid>.out* and the standard error stream (stderr) to the file *<pid>.err*. Note that *<pid>* here corresponds to the process id of the child process. As a result of this change you will not see any output from the child process on the terminal, you have to look at the corresponding *<pid>.out* or *<pid>.err* file for the corresponding output and error messages.
5. The parent process will make note of the time the program was started (you can use a timer such as the *time* function defined in *<time.h>* to capture the time before the fork method is invoked, you can find out more about *time* function by typing *man time*).
6. Then the parent process will wait for the child process to complete and when the child process terminates successfully, the parent process will capture the time the

child process completed (you can again use a timer function to capture the time when the *wait* function returns).
7. Open a log file (say, output.log) and write the command executed along with arguments, start time of the process, and the end time of the process separated by tabs. Use *ctime* function to write the time in a human readable form.
8. Go to step 2 and repeat the above process for very command in the input file.

Please use standard I/O library function calls for all I/O operations in this assignment. Make sure you open the file in the appropriate modes for reading and writing and also make sure to close the file and flush the file, especially when you are writing the file.

Here is a sample input file:

```
uname -a
/sbin/ifconfig
/home/UAB/puri/CS332/shared/hw2 500
/home/UAB/puri/CS332/shared/hw2 1000
```

Here is the corresponding sample output log file:

```
uname -a    Thu Oct 10 17:43:44 2019    Thu Oct 10 17:43:44 2019
/sbin/ifconfig    Thu Oct 10 17:43:44 2019    Thu Oct 10 17:43:44 2019
/home/UAB/puri/CS332/shared/hw2 500    Thu Oct 10 17:43:45 2019    Thu Oct 10
17:43:46 2019
/home/UAB/puri/CS332/shared/hw2 1000    Thu Oct 10 17:43:46 2019    Thu Oct 1
0 17:43:57 2019
```

You can download the C program `hw2.c` hw2.c writes to both stdout and stderr, you can use this program to test if the child process is writing stdout and stderr to the correct files. Compile the program using the following command:

```
$ gcc -Wall -O -o hw2 hw2.c
```