CS 332/532 – 1G- Systems Programming

Lab 12

Objectives

Use pipes to communicate between processes

Lab Assignment #12

Write a program what will prompt the user to enter a UNIX command that consists of all acceptable wildcards and special characters (characters a typical shell would accept) and execute the command using popen and pclose. The program should execute the command entered, display the output from the program to the terminal, and prompt the user to enter another command. When the user enters the command as "quit" the program should exit. Here is a sample session:

$ **./lab11**
Enter command: **ls *.c**
pager2.c pager.c pipe1.c pipe2a.c pipe2.c pipe3.c
Enter command: **ls -l pipe[1-3].c**
-rwxr-xr-x 1 puri domain users 2073 Oct 31 10:40 pipe1.c
-rwxr-xr-x 1 puri domain users 2784 Oct 31 10:40 pipe2.c
-rwxr-xr-x 1 puri domain users 3859 Oct 31 10:43 pipe3.c
Enter command: **ls pipe*.c | sort**
pipe1.c
pipe2a.c
pipe2.c
pipe3.c
Enter command: **ls pipe?.c | sort > files.out**
Enter command: **cat files.out**
pipe1.c
pipe2.c
pipe3.c
Enter command: **quit**
Exiting program...bye!
$

Assignment Submission

The assignment submission in Canvas is required. No late submissions will be accepted.

Submission Checklist:

- Upload the C source file (.c file) to Canvas as part of this lab submission. Submissions through the Canvas “Comments” will not be accepted.
- Upload a README.md file which should include:
- Instructions on how to compile your C source file into an executable.
- How to run the executable program.
- Any citation documentation.
- A link to your GitHub repository.

Please do not upload executables or object files. Independent Completion Forms are not required for labs.

Lab Workbook

Pipes

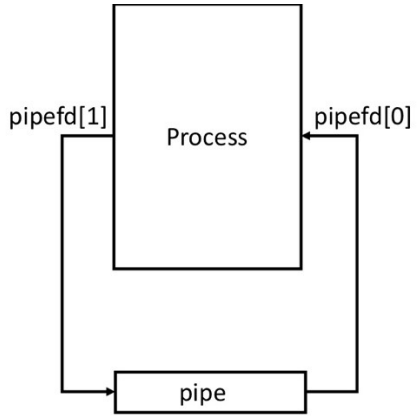
We have seen the Linux shell support pipes. For example:

```
$ ps -elf | grep ssh
```

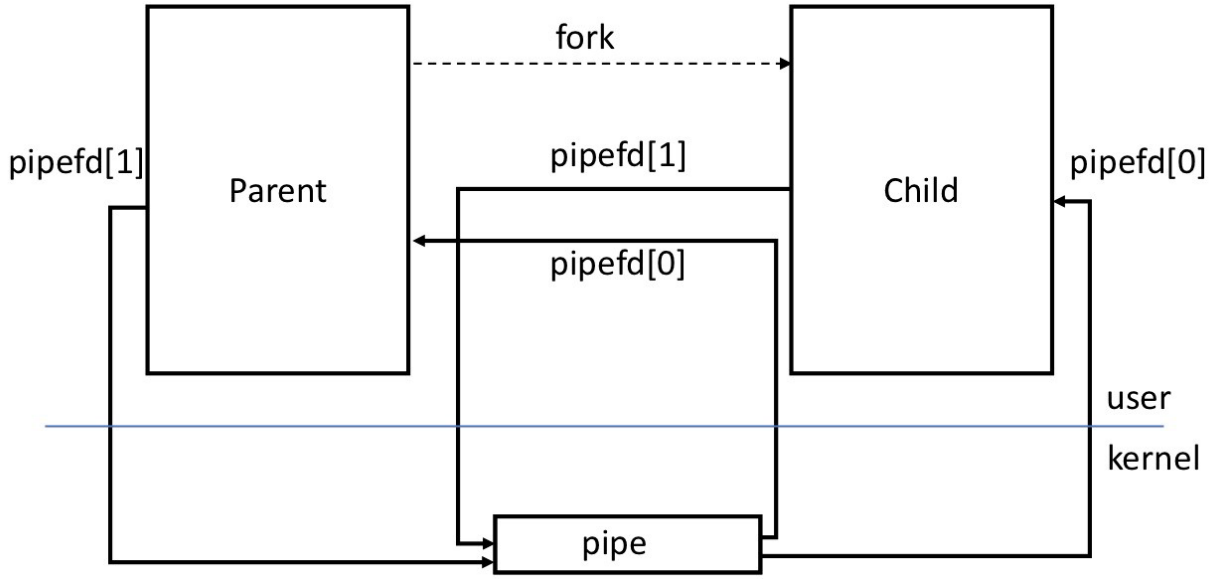
The above example redirects the output of the program *ps* to another program *grep* (instead of sending the output to standard output). Similarly, the program *grep* uses the output of *ps* as the input instead of a file name as the argument. The shell implements this redirection using pipes. The system call pipe is used to create a pipe and in most Linux systems pipes provide a **unidirectional flow** of data between two processes. The C API for the pipe function is shown below:

```
#include <unistd.h>
int pipe(int pipefd[2]);
```

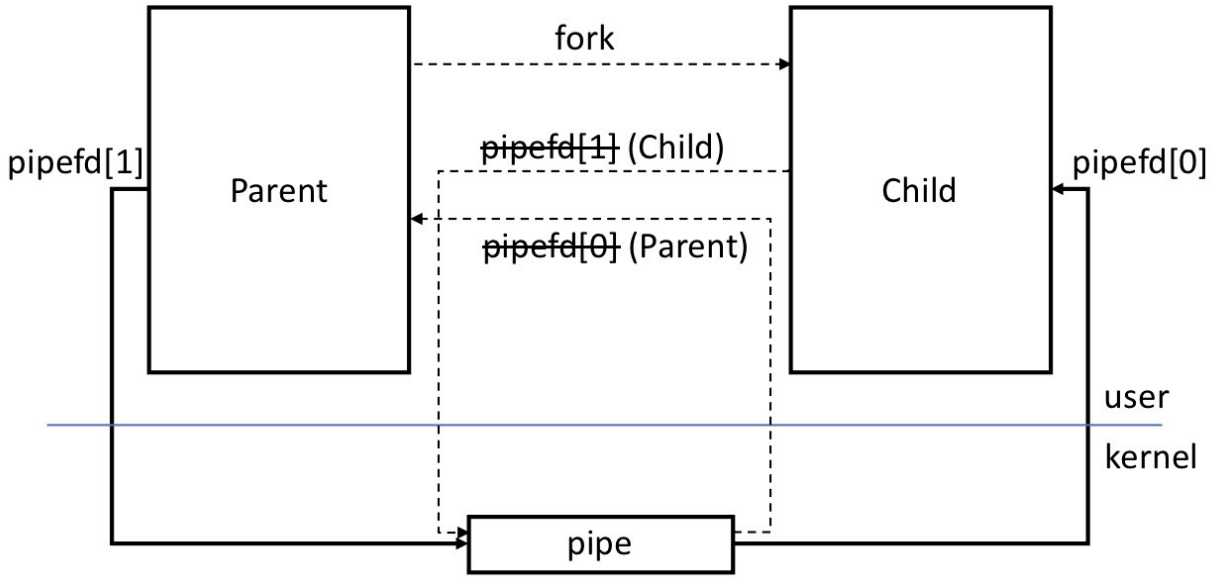
The pipe call returns two file descriptors corresponding to the read and write ends of the pipe. The first file descriptor (*pipefd[0]*) refers to the read end of the pipe (can be used for reading data from the pipe) and the second file descriptor (*pipefd[1]*) refers to the write end of the pipe (can be used for writing data to the pipe). The kernel buffers the data written to the pipe until it is read from the read end of the pipe. When there is an error in creating the pipe, it returns -1 and sets the corresponding *errno*, otherwise it returns 0 on success. The diagram below illustrates the creation of a pipe in a single process.



We really don't need to create a pipe to communicate within the same process, typically we use pipes to communicate between a parent process and a child process. In such a case, first a pipe is created by the parent process and then it creates a child process using the fork command. Since fork creates a copy of the parent process, the child process will also inherit the all open file descriptors and will have access to the pipe as shown in the diagram below.



To provide a unidirectional data channel for communication between the two process, the parent process closes the read end of the pipe and the child process closes the write end of the pipe as shown in the diagram below.

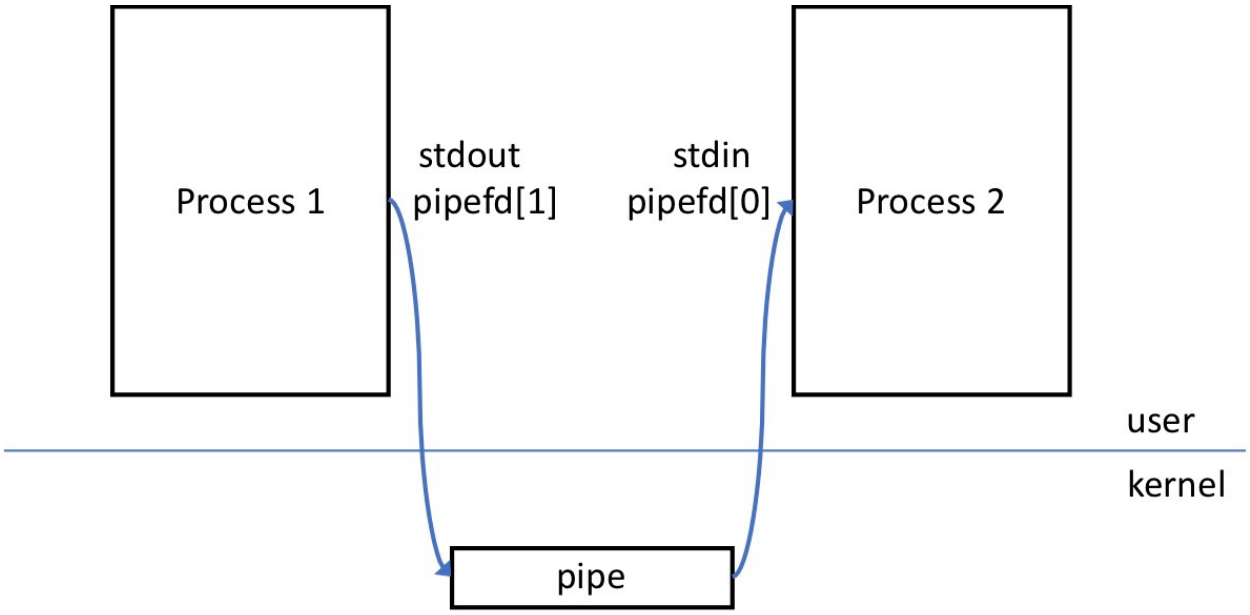


The following example shows the steps involved in creating a pipe, forking a child process, closing the file descriptors in the parent and child process, and communication between the parent and child process. The parent process writes the string passed as the command-line argument to the pipe and the child process reads the string from the pipe, converts the string to uppercase, and prints it to the standard output. The read and write functions that operate on files are used to read and write data from the pipe. Only part of the program is shown below, you can download the entire program here: pipe1.c

```
if (pipe(pipefd) == 0) { /* Open a pipe */
  if ((pid = fork()) == 0) { /* I am the child process */
    close(pipefd[1]);    /* close write end */

    while (read(pipefd[0], &c, 1) > 0) {
       c = toupper(c);
       write(1, &c, 1);
    }
    write(1, "\n", 1);
    close(pipefd[0]);

    exit(EXIT_SUCCESS);
  } else if (pid > 0) { /* I am the parent process */
    close(pipefd[0]);   /* close read end */

    write(pipefd[1], argv[1], strlen(argv[1]));
    close(pipefd[1]);

    wait(&status);      /* wait for child to terminate */
    if (WIFEXITED(status))
      printf("Child process exited with status = %d\n", WEXITSTATUS(status));
    else
      printf("Child process did not terminate normally!\n");
```

```
  } else { /* we have an error in fork */
    perror("fork");
    exit(EXIT_FAILURE);
  }
 } else {
  perror("pipe");
  exit(EXIT_FAILURE);
 }
```

The above example shows how the pipe is used by a parent and a child process to communicate. We can further extend this to implement pipes between any two programs such that the output of one program is redirected to the input of another program (e.g., ps -elf | grep ssh). In order to do this, we have to replace the standard output of the first program with the write end of the pipe and replace the standard input of the second program with the read end of the pipe. We have seen in the previous lab that this can be done using the dup2 system call. We will use the dup2 to perform this redirection and implement the pipe operation between two processes as shown in the diagram below.



We have several options to create the two processes, some of the possible options include:

1. The parent process creates a child process, the child process uses exec to launch the second program, and the parent process will use exec to launch the first program.
2. The parent process creates two child process, the first child process uses exec to launch the first program, the second child process uses exec to launch the second program, and the parent process waits for the two child processes to terminate.
3. The parent process create a child process, the child process creates another child process which in turn uses exec to launch the first program, then the child process uses exec to launch the second program, and the parent waits for the child process to terminate.

In this example, we will choose the second option and create two child processes to launch each program (you can find the solution using the first option here: pipe0.c. We follow the same approach as the example above and create a pipe in the parent process. First we create a child process, close the read end of the pipe, replace the standard output stream with the write end of the pipe using dup2 system call, and then use exec to launch the first program. Then we create another child process, close the write end of the pipe, replace the standard input stream with the read end of the pipe using dup2 system call, and then use exec to launch the second program. The parent then closes both ends of the pipe and uses the waitpid system call to wait for both child process to terminate. The name of the two programs to be executed is provided as command-line arguments to this program. The complete source code is available here: pipe2.c

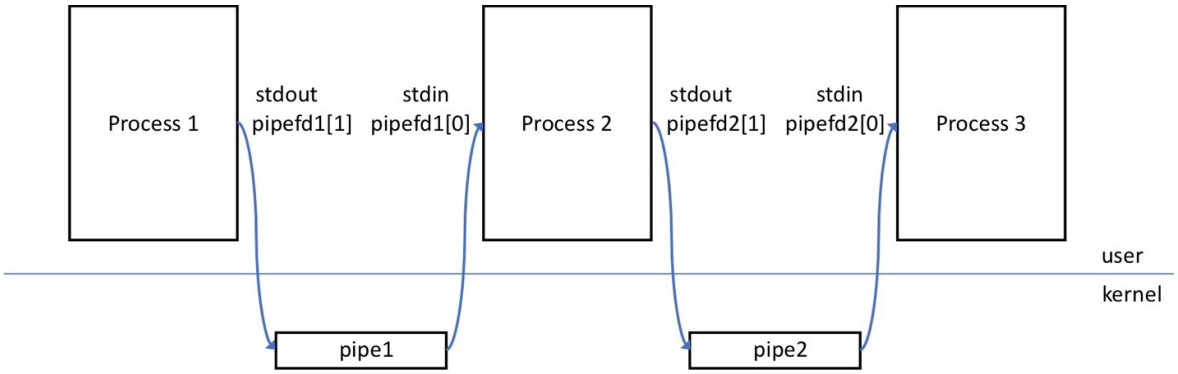
```
/* Simple program to illustrate how to implement pipes.
 * This example creates a pipe and then creates two child processes.
 * The first child replaces the stdout with write end of the pipe and
 * execs the first command. The second child replaces the stdin with
 * read end of the pipe and execs the second command. Thus, the stdout
 * of the first child is sent to the stdin of the second child. The parent
 * process closes both ends of the pipe and waits for the child process
 * to terminate.
 *
 * To Compile: gcc -Wall pipe2.c
 * To Run: ./a.out <command1> <command2>
 *         Output of <command1> will be the input for <command2>
 */

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

int main(int argc, char **argv) {
   pid_t pid1, pid2;
   int pipefd[2]; /* fildes[0] for read, fildes[1] for write */
   int status1, status2;

   if (argc != 3) {
      printf("Usage: %s <command1> <command2>\n", argv[0]);
      exit(EXIT_FAILURE);
   }

   if (pipe(pipefd) == 0) { /* Open a pipe */

     pid1 = fork(); /* fork first process to execute command1 */
     if (pid1 == 0) { /* this is the child process */
       /* close read end of the pipe */
               close(pipefd[0]);

               /* replace stdout with write end of pipe */
```

```
            if (dup2(pipefd[1], 1) == -1) {
        perror("dup2");
        exit(EXIT_FAILURE);
      }

      /* execute <command1> */
      execlp(argv[1], argv[1], (char *)NULL);
      printf("If you see this statement then exec failed ;-(\n");
      perror("execlp");
      exit(EXIT_FAILURE);

    } else if (pid1 < 0) { /* we have an error */
      perror("fork"); /* use perror to print the system error message */
      exit(EXIT_FAILURE);
    }

    pid2 = fork(); /* fork second process to execute command2 */
    if (pid2 == 0) { /* this is child process */
      /* close write end of the pipe */
            close(pipefd[1]);

      /* replace stdin with read end of pipe */
            if (dup2(pipefd[0], 0) == -1) {
        perror("dup2");
        exit(EXIT_FAILURE);
      }

      /* execute <command2> */
      execlp(argv[2], argv[2], (char *)NULL);
      printf("If you see this statement then exec failed ;-(\n");
      perror("execlp");
      exit(EXIT_FAILURE);

    } else if (pid2 < 0) { /* we have an error */
      perror("fork"); /* use perror to print the system error message */
      exit(EXIT_FAILURE);
    }

    /* close the pipe in the parent */
    close(pipefd[0]);
    close(pipefd[1]);

    /* wait for both child processes to terminate */
    waitpid(pid1, &status1, 0);
    waitpid(pid2, &status2, 0);
  } else {
    perror("pipe");
    exit(EXIT_FAILURE);
  }

  return 0;
}
```

You can compile and test this program as follows (note that this program does not take any arguments to the two commands passed, e.g., ls -l or wc -l):

```
$ ./a.out ls sort
.....
$ ./a.out ls wc
.....
```

We can extend this to three processes if we like to implement something like: *ls | sort | wc.* We will create three processes and use two pipes - one for communication between the first and second process and one for communication between second and third process. The code for the first and third children will be similar to the example above while the second child has to replace both standard input and standard output streams instead of just one. The diagram below illustrates this and the complete source code is available here: pipe3.c



Additional example

A simpler version of the program in Figure 15.6 from the textbook: pager.c

popen and pclose functions

As we have seen in the section above, the common usage of pipes involve creating a pipe, creating a child process with fork, closing the unused ends of the pipe, execing a command in the child process, and waiting for the child process to terminate in the parent process. Since this is such a common usage, UNIX systems provide *popen* and *pclose* functions that perform most of these operations in a single operation. The C APIs for the *popen* and *pclose* functions are shown below:

```
#include <stdio.h>

FILE *popen(const char *command, const char *type);
int pclose(FILE *stream);
```

The popen function performs the following steps:

- creates a pipe

- creates a new process using fork
- perform the following steps in the child process
    - close unused ends of the pipe (based on the *type* argument)
    - execs a shell to execute the *command* provided as argument to popen (i.e., executes "sh -c command")
- perform the following steps in the parent process
    - close unused ends of the pipe (based on the *type* argument)
    - wait for the child process to terminate

The *popen* function returns the FILE handle to the pipe created so that the calling process can read or write to the pipe using standard I/O system calls. If the *type* argument is specified as read-only ("r") then the calling process can read from the pipe, this results in reading from the *stdout* of the child process (see Figure 15.9). If the type argument is specifies as write-only ("w") then the calling process can write to the pipe, this results in writing to the *stdin* of the child process created (see Figure 15.10).

The FILE handle returned by *popen* must be closed using *pclose* to make sure that the I/O stream opened to read or write to the pipe is closed and wait for the child process to terminate. The termination status of the shell started by exec will be returned when the *pclose* function returns.

Here is the source code for the pipe2.c example using popen and pclose. The complete source code is available here: pipe2a.c

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

int main(int argc, char **argv) {
   FILE *fp1, *fp2;
   char line[BUFSIZ];

   if (argc != 3) {
      printf("Usage: %s <command1> <command2>\n", argv[0]);
      exit(EXIT_FAILURE);
   }

   /* create a pipe, fork/exec command argv[1], in "read" mode */
   /* read mode - parent process reads stdout of child process */
   if ((fp1 = popen(argv[1], "r")) == NULL) {
               perror("popen");
               exit(EXIT_FAILURE);
   }

   /* create a pipe, fork/exec command argv[2], in "write" mode */
   /* write mode - parent process writes to stdin of child process */
   if ((fp2 = popen(argv[2], "w")) == NULL) {
               perror("popen");
               exit(EXIT_FAILURE);
   }
```

```
   /* read stdout from child process 1 and write to stdin of
      child process 2 */
   while (fgets(line, BUFSIZ, fp1) != NULL) {
               if (fputs(line, fp2) == EOF) {
                  printf("Error writing to pipe\n");
                  exit(EXIT_FAILURE);
      }
   }

   /* wait for child process to terminate */
   if ((pclose(fp1) == -1) || pclose(fp2) == -1) {
               perror("pclose");
               exit(EXIT_FAILURE);
   }

   return 0;
}
```

Here is sample output when the new version of the program is executed:

```
$ ./a.out ls wc
     7      7     56
$ ls | wc
     7      7     56
$ ./a.out "ps -el" "grep ssh"
4 S     0 16691     1  0  80   0 - 28230 poll_s ?       00:00:14 sshd
4 S     0 18801 16691  0  80   0 - 49947 poll_s ?       00:00:00 sshd
5 S 50927 18813 18801  0  80   0 - 49947 poll_s ?       00:00:00 sshd
$ ps -el | grep ssh
4 S     0 16691     1  0  80   0 - 28230 poll_s ?       00:00:14 sshd
4 S     0 18801 16691  0  80   0 - 49947 poll_s ?       00:00:00 sshd
5 S 50927 18813 18801  0  80   0 - 49947 poll_s ?       00:00:00 sshd
```

Note that since the command is executed using a shell, we can provide wildcards and other special characters that the shell can expand. Also note that in this version of the program the parent process is reading the *stdout* stream of the first child process and then writing to the *stdin* stream of the second child process (we did not do this in the first version). What is the reason for this change in this version of the program?

Here is an updated version of the pager program that uses popen and pclose: pager2.c

You can also find a simpler version of the program that uses a single popen system call to create a pipe in "read" mode, execute the command specified as the command-line argument, reads the pipe and prints it to stdout: popen.c