Lab Exercise #6 -- UNIX Files and Pipes

This exercise focuses on the system calls and library functions available under the Solaris and Linux operating systems to manipulate files and pipes.

A. UNIX Files and File Descriptors

In UNIX, I/O streams are identified within processes using file descriptors, which are unsigned integers mapped to specific files. By default, each process begins execution with three valid file descriptors:

```
0 -- stdin (standard input stream)
1 -- stdout (standard output stream)
2 -- stderr (standard error stream)
```

System calls normally use file descriptors, while library functions normally use symbolic names which are associated with file descriptors. For example:

```
// Copy 128 bytes from "buffer" to the standard output stream

write( 1, buffer, 128 );

// Copy the string "HELP!" to the standard error stream

fprintf( stderr, "HELP!\n" );
```

Most programs use calls to library functions (such as "scanf" and "printf") to manipulate files. Of course, those library functions then use the appropriate system calls to accomplish the task.

The biggest difference between the low-level I/O functions (such as "read" and "write") and the C library functions (such as "scanf" and "printf") is that the low-level functions simply transfer bytes without attempting to interpret them or convert them through formatting specifications.

1) Examine the program in "lab06.copy1.c", then translate and execute it.

a) What is the effect of executing the program?

b) How is "end-of-file" detected by the program?

c) Describe the arguments used in the first call to "open".

d) Describe the arguments used in the second call to "open".

2) Examine the program in "lab06.copy2.c" (a more robust version of the "copy" program), then translate and execute it.

a) Under what circumstances will the first call to "open" fail?

b) Under what circumstances will the second call to "open" fail?

c) Copy the program into your account and make the following modifications:

-- display the number of bytes copied to "out_fd" by each call to "write"

-- display the total number of bytes copied to "out_fd" by the program

B. UNIX Pipelines

Pipeliness are the oldest form of UNIX interprocess communication; they are used to establish one-way communication between two processes that share a common ancestor. The "pipe" system call is used to create a pipeline:

```
int pipe( int fd[2] );
```

Two file descriptors are returned through the argument: "fd[0]" is open for reading and "fd[1]" is open for writing.

Typically, a process creates the pipeline, then uses "fork" to create a child process. Each process now has a copy of the file descriptor array; one process writes data into the pipeline, while the other reads from it.

1) Examine the program in "lab06.pipe1.c", then translate and execute it.

a) What is the effect of executing the program?

b) Execute the program several times. Is the order in which the output appears the same in all cases? Explain.

2) Copy the program into your account and conduct the following experiments.

a) Modify the parent process so that it pauses for 2 seconds (by calling "sleep") after writing into the pipeline, but before closing the "write" end of the pipeline. How does this modification effect the output? Explain.

b) Modify the parent process so that it goes into an (empty) infinite loop after writing into the pipeline, but before closing the "write" end of the pipeline. How does this modification effect the output? Explain.

c) How does the child process detect "end-of-file"? Explain.

d) What would happen if the parent process never closed the "write" end of the pipeline? Explain.

3) Examine the program in "lab06.pipe2.c", then translate and execute it.

a) What is the effect of executing the program?

b) What is the purpose of the "dup2" function? Explain.

4) Copy the program into your account and modify it so that the program creates two separate child processes, one of which executes "ps" and one of which executes "grep" (the output of "ps" should be still be piped to "grep"). The parent process should display a message after both children terminate.