ENCE360 Lab 3 - Signals, Pipes, and File Descriptors

1. Objectives

The overall goal of this lab is to introduce two forms of communication: signals and pipes. You will have to read, compile and run some small applications written in C for Linux. During this lab, you should enhance your understanding of the concepts of pipes and signals by implementing C programs that demonstrate both.

2. Preparation

Download the "Lab 3 pipes" zip archive from Learn for a copy of the files to be used in this laboratory: `dup2eg.c`, `pipe.c`, `pipedup2.c` and `waiting.c` should now be in the current directory.

3. Child Process termination

Once we have created a child process using `fork()`, there are two possibilities. Either the parent process exits before the child, or the child exits before the parent. Now, Unix's semantics regarding parent-child process relations state something like this:

When a child process exits, it is not immediately cleared off the process table. Instead, a signal is sent to its parent process, which needs to acknowledge it's child's death, and only then the child process is completely removed from the system. In the duration before the parent's acknowledgment and after the child's exit, the child process is in a state called 'zombie'.

When the parent process is not properly coded, the child remains in the zombie state forever. Such processes can be noticed by running the 'ps' command (shows the process list), and seeing processes having the string '<defunct>' as their command name.

The wait() System Call

The simple way of a process to acknowledge the death of a child process is by using the `wait()` system call. When `wait()` is called, the process is suspended until one of its child processes exits, and then the call returns with the exit status of the child process. If it has a zombie child process, the call returns immediately, with the exit status of that process.

Asynchronous Child Death Notification.

The problem with calling `wait()` directly, is that usually you want the parent process to do other things, while its child process executes its code. That problem has a solution by using signals. When a child process dies, a signal, SIGCHLD (or SIGCLD) is sent to its parent process. Thus, using a proper signal handler, the parent will get an asynchronous notification, and then when it calls `wait()`, the system assures that the call will return immediately, since there is already a zombie child.

4. Signals

Signals are a primitive way of communicating information between processes. They occur asynchronously (with no specified timing or sequencing). A list of signal constants, their numeric values, and a description of their meanings is accessed by entering the command `man 7 signal`.
These are defined in the system include file `signal.h`.

4.1. To do

- Compile and execute `waiting.c`.
- Copy `waiting.c` to `asyncWaiting.c`. Modify `asyncWaiting.c` so that instead of immediately waiting for the child process after forking, the waiting is delayed until the SIGCHLD signal is caught by the parent process. This involves:
    - Defining a signal handler, the function `void waitChild(int sigNum)`, which will call `wait()` when the parent receives the SIGCHLD signal.
    - The parent registers this signal handler using the system call `signal()` before the child is created. This registration process is described in more detail by the online manual entry for signal (man signal).

Observe and understand the differences in the output produced from `waiting.c` and `asyncWaiting.c`.

5. Pipes

The program pipe.c creates a pipe, then forks, and the parent process sends the first command line argument via the created pipe to the child.

5.1. To do

- Compile and execute `pipe.c`
- Copy `pipe.c` to `pipeToUpper.c`. Modify `pipeToUpper.c` so that the child changes the received message to upper case using the `toupper` library call and sends the message back to the parent. See the online manual entry for details about `toupper`.

Assume pipes are one-way (half duplex). Your program then should have two pipes, one for child →parent communication, and one for parent --> child communication.

6. Duplicating file descriptors

File descriptors can be duplicated. For example, when you type `ls > my.file` in the shell, the shell:

1. Opens `my.file`,
2. forks a child.
3. The child process then calls the `dup2` system call to close the `stdout` file descriptor (which usually points to a (pseudo)terminal, such as one found in an `xterm` window), and replace it with the `my.file` file descriptor.

4. `dup2` then closes the old `my.file` file descriptor. Now, any writes to `stdout` will go to `my.file` instead of the terminal.

5. Finally, the shell execs the `ls` command. The `ls` command just writes to `stdout` as usual, and the output goes to `my.file`.

`dup2eg.c` is a file which illustrates the example described above (without the fork).

6.1. To do

- Compile and execute `dup2eg.c`.
  Copy `dup2eg.c` to `dup2sort.c`. Modify `dup2sort.c` to execute the sort program. Give the sort program the argument '-k +7' so that the listing is sorted by time and set up `sort's stdin` to come from a file called `my.file`.

7. The dup2 call with pipes

Because each end of a pipe is a file descriptor, dup2 works with pipes also. For example, when you type `ls -l |sort -k +8`, the shell does something like this:

1. The shell creates a pipe using the pipe system call.
2. The shell forks a child. The child gets a copy of the parent's file descriptor table.
3. The child uses `dup2` to close its existing `stdout`, and replace it with the write-end of the pipe. The file descriptors for the read and write end of the pipe are then closed. (The pipe is still in existence, just these pointers to the ends are closed. `stdout` is pointing to the write-end).
4. The shell forks another child. This child does a similar thing to the first child, except with `stdin` and the read-end of the pipe.
5. The first child execs the `ls` command, and the second child execs the sort command.

There is an example of this in the file `pipedup2.c`, without the extra fork that the shell does for the second child.

7.1. To do

1. Compile and execute `pipedup2.c`.

2. Copy `pipedup2.c` to `myPipeDup2.c`. Modify `myPipeDup2.c` so that the output of the sort command is piped to `head -5`. The program now will be the same as the shell command line `ls -l |sort -k +9 |head -5`.

Note that you ought to include the full path name to the `head`, `sort` and `ls` commands. Thus, the commands are `/bin/ls`, `/bin/sort`, (`/usr/bin/sort` on some distributions) and `/usr/bin/head` respectively.