Next | Up | Previous | Contents

Next: 6.2.3 Pipes the Easy **Up:** 6.2 Half-duplex UNIX Pipes **Previous:** 6.2.1 Basic Concepts

6.2.2 Creating Pipes in C

Creating ``pipelines'' with the C programming language can be a bit more involved than our simple shell example. To create a simple pipe with C, we make use of the pipe() system call. It takes a single argument, which is an array of two integers, and if successful, the array will contain two new file descriptors to be used for the pipeline. After creating a pipe, the process typically spawns a new process (remember the child inherits open file descriptors).

```
  SYSTEM CALL: pipe();

  PROTOTYPE: int pipe( int fd[2] );
    RETURNS: 0 on success
             -1 on error: errno = EMFILE (no free descriptors)
                                  EMFILE (system file table is full)
                                  EFAULT (fd array is not valid)

  NOTES: fd[0] is set up for reading, fd[1] is set up for writing
```

The first integer in the array (element 0) is set up and opened for reading, while the second integer (element 1) is set up and opened for writing. Visually speaking, the output of fd1 becomes the input for fd0. Once again, all data traveling through the pipe moves through the kernel.

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

        main()
        {
                int     fd[2];

                pipe(fd);
                .
                .
        }
```

Remember that an array name in C *decays* into a pointer to its first member. Above, `fd` is equivalent to `&fd[0]`. Once we have established the pipeline, we then fork our new child process:

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

```
#include <sys/types.h>

main()
{
        int     fd[2];
        pid_t   childpid;

        pipe(fd);

        if((childpid = fork()) == -1)
        {
                perror("fork");
                exit(1);
        }
        .
        .
}
```

If the parent wants to receive data from the child, it should close fd1, and the child should close fd0. If the parent wants to send data to the child, it should close fd0, and the child should close fd1. Since descriptors are shared between the parent and child, we should always be sure to close the end of pipe we aren't concerned with. On a technical note, the EOF will never be returned if the unnecessary ends of the pipe are not explicitly closed.

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

main()
{
        int     fd[2];
        pid_t   childpid;

        pipe(fd);

        if((childpid = fork()) == -1)
        {
                perror("fork");
                exit(1);
        }

        if(childpid == 0)
        {
                /* Child process closes up input side of pipe */
                close(fd[0]);
        }
        else
        {
                /* Parent process closes up output side of pipe */
                close(fd[1]);
        }
        .
```

```
                .
        }
```

As mentioned previously, once the pipeline has been established, the file descriptors may be treated like descriptors to normal files.

```
/*****************************************************************************
 Excerpt from "Linux Programmer's Guide - Chapter 6"
 (C)opyright 1994-1995, Scott Burkett
 *****************************************************************************
 MODULE: pipe.c
 *****************************************************************************/

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

int main(void)
{
        int     fd[2], nbytes;
        pid_t   childpid;
        char    string[] = "Hello, world!\n";
        char    readbuffer[80];

        pipe(fd);

        if((childpid = fork()) == -1)
        {
                perror("fork");
                exit(1);
        }

        if(childpid == 0)
        {
                /* Child process closes up input side of pipe */
                close(fd[0]);

                /* Send "string" through the output side of pipe */
                write(fd[1], string, (strlen(string)+1));
                exit(0);
        }
        else
        {
                /* Parent process closes up output side of pipe */
                close(fd[1]);

                /* Read in a string from the pipe */
                nbytes = read(fd[0], readbuffer, sizeof(readbuffer));
                printf("Received string: %s", readbuffer);
        }

        return(0);
}
```

Often, the descriptors in the child are duplicated onto standard input or output. The child can then exec() another program, which inherits the standard streams. Let's look at the dup() system call:

```
SYSTEM CALL: dup();

PROTOTYPE: int dup( int oldfd );
  RETURNS: new descriptor on success
           -1 on error: errno = EBADF (oldfd is not a valid descriptor)
                                EBADF (newfd is out of range)
                                EMFILE (too many descriptors for the process)

NOTES: the old descriptor is not closed!  Both may be used interchangeably
```

Although the old descriptor and the newly created descriptor can be used interchangeably, we will typically close one of the standard streams first. The dup() system call uses the lowest-numbered, unused descriptor for the new one.

Consider:

```
        .
        .
        childpid = fork();

        if(childpid == 0)
        {
                /* Close up standard input of the child */
                close(0);

                /* Duplicate the input side of pipe to stdin */
                dup(fd[0]);
                execlp("sort", "sort", NULL);
                .
        }
```

Since file descriptor 0 (stdin) was closed, the call to dup() duplicated the input descriptor of the pipe (fd0) onto its standard input. We then make a call to execlp(), to overlay the child's text segment (code) with that of the sort program. Since newly exec'd programs inherit standard streams from their spawners, it actually inherits the input side of the pipe as its standard input! Now, anything that the original parent process sends to the pipe, goes into the sort facility.

There is another system call, dup2(), which can be used as well. This particular call originated with Version 7 of UNIX, and was carried on through the BSD releases and is now required by the POSIX standard.

```
SYSTEM CALL: dup2();
```

```
  PROTOTYPE: int dup2( int oldfd, int newfd );
    RETURNS: new descriptor on success
             -1 on error: errno = EBADF (oldfd is not a valid descriptor)
                                  EBADF (newfd is out of range)
                                  EMFILE (too many descriptors for the process)

  NOTES: the old descriptor is closed with dup2()!
```

With this particular call, we have the close operation, and the actual descriptor duplication, wrapped up in one system call. In addition, it is guaranteed to be atomic, which essentially means that it will never be interrupted by an arriving signal. The entire operation will transpire before returning control to the kernel for signal dispatching. With the original dup() system call, programmers had to perform a close() operation before calling it. That resulted in two system calls, with a small degree of vulnerability in the brief amount of time which elapsed between them. If a signal arrived during that brief instance, the descriptor duplication would fail. Of course, dup2() solves this problem for us.

Consider:

```
        .
        .
        childpid = fork();

        if(childpid == 0)
        {
                /* Close stdin, duplicate the input side of pipe to stdin */
                dup2(0, fd[0]);
                execlp("sort", "sort", NULL);
                .
                .
        }
```

Next Up Previous Contents

Next: 6.2.3 Pipes the Easy **Up:** 6.2 Half-duplex UNIX Pipes **Previous:** 6.2.1 Basic Concepts

Converted on:
Fri Mar 29 14:43:04 EST 1996