Introduction to Inter-process Communication Using Named Pipes

Introduction

Named pipes allow two unrelated processes to communicate with each other. They are also known as FIFOs (first-in, first-out) and can be used to establish a one-way (half-duplex) flow of data.

Named pipes are identified by their access point, which is basically in a file kept on the file system. Because named pipes have the pathname of a file associated with them, it is possible for unrelated processes to communicate with each other; in other words, two unrelated processes can open the file associated with the named pipe and begin communication. Unlike anonymous pipes, which are process-persistent objects, named pipes are file system-persistent objects, that is, they exist beyond the life of the process. They have to be explicitly deleted by one of the processes by calling "unlink" or else deleted from the file system via the command line.

In order to communicate by means of a named pipe, the processes have to open the file associated with the named pipe. By opening the file for reading, the process has access to the reading end of the pipe, and by opening the file for writing, the process has access to the writing end of the pipe.

A named pipe supports blocked read and write operations by default: if a process opens the file for reading, it is blocked until another process opens the file for writing, and vice versa. However, it is possible to make named pipes support non-blocking operations by specifying the O_NONBLOCK flag while opening them. A named pipe must be opened either read-only or write-only. It must not be opened for read-write because it is half-duplex, that is, a one-way channel.

Shells make extensive use of pipes; for example, we use pipes to send the output of one command as the input of the other command. In real-life UNIX® applications, named pipes are used for communication, when the two processes need a simple method for synchronous communication.

Creating a Named Pipe

A named pipe can be created in two ways -- via the command line or from within a program.

From the Command Line

A named pipe may be created from the shell command line. For this one can use either the "`mknod`" or "`mkfifo`" commands.

Example:

To create a named pipe with the file named "npipe" you can use one of the following commands:

```
% mknod npipe p
```

or

```
% mkfifo npipe
```

You can also provide an absolute path of the named pipe to be created.

Now if you look at the file using "ls ?l", you will see the following output:

```
prw-rw-r-- 1 secf other   0  Jun 6 17:35  npipe
```

The 'p' on the first column denotes that this is a named pipe. Just like any file in the system, it has access permissions that define which users may open the named pipe, and whether for reading, writing, or both.

Within a Program

The function "`mkfifo`" can be used to create a named pipe from within a program. The signature of the function is as follows:

```
int mkfifo(const char *path, mode_t mode)
```

The `mkfifo` function takes the path of the file and the mode (permissions) with which the file should be created. It creates the new named pipe file as specified by the path.

The function call assumes the O_CREATE|O_EXCL flags, that is, it creates a new named pipe or returns an error of EEXIST if the named pipe already exists. The named pipe's owner ID is set to the process' effective user ID, and its group ID is set to the process' effective group ID, or if the S_ISGID bit is set in the parent directory, the group ID of the named pipe is inherited from the parent directory.

Opening a Named Pipe

A named pipe can be opened for reading or writing, and it is handled just like any other normal file in the system. For example, a named pipe can be opened by using the `open()` system call, or by using the `fopen()` standard C library function.

As with normal files, if the call succeeds, you will get a file descriptor in the case of `open()`, or a 'FILE' structure pointer in the case of `fopen()`, which you may use either for reading or for writing, depending on the parameters passed to `open()` or to `fopen()`.

Therefore, from a user's point of view, once you have created the named pipe, you can treat it as a file so far as the operations for opening, reading, writing, and deleting are concerned.

Reading From and Writing to a Named Pipe

Reading from and writing to a named pipe are very similar to reading and writing from or to a normal file. The standard C library function calls `read( )` and `write( )` can be used for reading from and writing to a named pipe. These operations are blocking, by default.

The following points need to be kept in mind while doing read/writes to a named pipe:

A named pipe cannot be opened for both reading and writing. The process opening it must choose either read mode or write mode. The pipe opened in one mode will remain in that mode until it is closed.
Read and write operations to a named pipe are blocking, by default. Therefore if a process reads from a named pipe and if the pipe does not have data in it, the reading process will be blocked. Similarly if a process tries to write to a named pipe that has no reader, the writing process gets blocked, until another process opens the named pipe for reading. This, of course, can be overridden by specifying the O_NONBLOCK flag while opening the named pipe.
Seek operations (via the Standard C library function lseek) cannot be performed on named pipes.

Full-Duplex Communication Using Named Pipes

Although named pipes give a half-duplex (one-way) flow of data, you can establish full-duplex communication by using two different named pipes, so each named pipe provides the flow of data in one direction. However, you have to be very careful about the order in which these pipes are opened in the client and server, otherwise a deadlock may occur.

For example, let us say you create the following named pipes:

NP1 and NP2

In order to establish a full-duplex channel, here is how the server and the client should treat these two named pipes:

Let us assume that the server opens the named pipe NP1 for reading and the second pipe NP2 for writing. Then in order to ensure that this works correctly, the client must open the first named pipe NP1 for writing and the second named pipe NP2 for reading. This way a full-duplex channel can be established between the two processes.

Failure to observe the above-mentioned sequence may result in a deadlock situation.

Benefits of Named Pipes

- Named pipes are very simple to use.
- `mkfifo` is a thread-safe function.
- No synchronization mechanism is needed when using named pipes.
- Write (using `write` function call) to a named pipe is guaranteed to be atomic. It is atomic even if the named pipe is opened in non-blocking mode.
- Named pipes have permissions (read and write) associated with them, unlike anonymous pipes. These permissions can be used to enforce secure communication.

Limitations of Named Pipes

- Named pipes can only be used for communication among processes on the same host machine.
- Named pipes can be created only in the local file system of the host, that is, you cannot create a named pipe on the NFS file system.
- Due to their basic blocking nature of pipes, careful programming is required for the client and server, in order to avoid deadlocks.
- Named pipe data is a byte stream, and no record identification exists.

Code Samples

The code samples given here were compiled using the GNU C compiler version 3.0.3 and were run and tested on a SPARC processor-based Sun Ultra 10 workstation running the Solaris 8 Operating Environment.

The following code samples illustrate half-duplex and full-duplex communication between two unrelated processes by using named pipes.

Example of Half-Duplex Communication

In the following example, a client and server use named pipes for one-way communication. The server creates a named pipe, opens it for reading and waits for input on the read end of the pipe. Named-pipe reads are blocking by default, so the server waits for the client to send some request on the pipe. Once data becomes available, it converts the string to upper case and prints via STDOUT.

Figure 1: Half-Duplex Communication

The client opens the same named pipe in write mode and writes a user-specified string to the pipe (see Figure 1).

The following table shows the contents of the header file used by both the client and server. It contains the definition of the named pipe that is used to communicate between the client and the server.

Filename : half_duplex.h

```
#define HALF_DUPLEX            "/tmp/halfduplex"
#define MAX_BUF_SIZE>  255
```

Server Code

The following table shows the contents of `Filename : hd_server.c`.

```
#include <stdio.h>
#include <errno.h>
#include <ctype.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <halfduplex.h> /* For name of the named-pipe */

int main(int argc, char *argv[])
{
    int fd, ret_val, count, numread;
    char buf[MAX_BUF_SIZE];

    /* Create the named - pipe */
    ret_val = mkfifo(HALF_DUPLEX, 0666);

    if ((ret_val == -1) && (errno != EEXIST)) {
        perror("Error creating the named pipe");
        exit (1);
    }

    /* Open the pipe for reading */
    fd = open(HALF_DUPLEX, O_RDONLY);

    /* Read from the pipe */
    numread = read(fd, buf, MAX_BUF_SIZE);

    buf[numread] = '0';

    printf("Half Duplex Server : Read From the
    pipe : %sn", buf);

    /* Convert to the string to upper case */
    count = 0;
    while (count < numread) {
        buf[count] = toupper(buf[count]);
        count++;
    }

    printf("Half Duplex Server : Converted String : %sn", buf);
}
```

Client Code

The following table shows the contents of `Filename : hd_client.c`.

```
#include <stdio.h>
#include <errno.h>
#include <ctype.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <halfduplex.h> /* For name of the named-pipe */

int main(int argc, char *argv[])
{
    int fd;

    /* Check if an argument was specified. */

    if (argc != 2) {
        printf("Usage : %s <string to be sent to the server>n", argv[0]);
        exit (1);
    }

    /* Open the pipe for writing */
    fd = open(HALF_DUPLEX, O_WRONLY);

    /* Write to the pipe */
    write(fd, argv[1], strlen(argv[1]));
}
```

Running the Client and the Server

When you run the server, it will block on the read call and will wait until the client writes something to the named pipe. After that it will print what it read from the pipe, convert the string to upper case, and then terminate. In a typical implementation this server will be either an iterative or a concurrent server. But for simplicity and to demonstrate the communication through the named pipe, we have kept the server code very simple. When you run the client, you will need to give a string as an argument.

Make sure you run the server first, so that the named pipe gets created.

Expected output:

1. Run the server:

   ```
   % hd_server &
   ```

   The server program will block here, and the shell will return control to the command line.

2. Run the client:

   ```
   % hd_client hello
   ```

3. The server prints the string read and terminates:

   ```
   Half Duplex Server : Read From the pipe : hello
   Half Duplex Server : Converted String : HELLO
   ```

Example of Full-Duplex Communication

In the following example, a client and server use named pipes for two-way communication. The server creates two named pipes. It opens the first pipe for reading and the second pipe for writing to communicate back to the client. It then waits for input on the read pipe. Once data is available, it converts the string to upper case and writes the converted string to the write pipe, which the client will read and print.

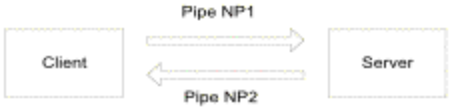

Figure 2: Full-Duplex Communication

The client opens the first pipe for writing, and it sends data through this pipe to the server. The client opens the second pipe for reading, and through this pipe, it reads the server's response (see Figure 2).

The following table shows the contents of the header file used by both the client and server. It contains the definition of the two named pipes that are used to communicate between the client and the server.

Filename : fullduplex.h

```
#define NP1            "/tmp/np1"
#define NP2            "/tmp/np2"
#define MAX_BUF_SIZE   255
```

Server Code

The following table shows the contents of `Filename : fd_server.c`.

```
#include <stdio.h>
#include <errno.h>
#include <ctype.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <fullduplex.h> /* For name of the named-pipe */

int main(int argc, char *argv[])
{
    int rdfd, wrfd, ret_val, count, numread;
    char buf[MAX_BUF_SIZE];

    /* Create the first named - pipe */
    ret_val = mkfifo(NP1, 0666);

    if ((ret_val == -1) && (errno != EEXIST)) {
        perror("Error creating the named pipe");
        exit (1);
    }

    ret_val = mkfifo(NP2, 0666);

    if ((ret_val == -1) && (errno != EEXIST)) {
        perror("Error creating the named pipe");
        exit (1);
    }

    /* Open the first named pipe for reading */
    rdfd = open(NP1, O_RDONLY);

    /* Open the second named pipe for writing */
    wrfd = open(NP2, O_WRONLY);

    /* Read from the first pipe */
    numread = read(rdfd, buf, MAX_BUF_SIZE);

    buf[numread] = '0';

    printf("Full Duplex Server : Read From the
    pipe : %sn", buf);

    /* Convert to the string to upper case */
    count = 0;
    while (count < numread) {
        buf[count] = toupper(buf[count]);
        count++;
    }

    /*
     * Write the converted string back to the second
     * pipe
     */
    write(wrfd, buf, strlen(buf));
}
```

Client Code

The following table shows the contents of `Filename : hd_client.c`.

```
#include <stdio.h>
#include <errno.h>
#include <ctype.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <fullduplex.h> /* For name of the named-pipe */

int main(int argc, char *argv[])
{
    int wrfd, rdfd, numread;
    char rdbuf[MAX_BUF_SIZE];

    /* Check if an argument was specified. */

    if (argc != 2) {
        printf("Usage : %s <string to be sent to the server>n", argv[0]);
        exit (1);
    }

    /* Open the first named pipe for writing */
    wrfd = open(NP1, O_WRONLY);

    /* Open the second named pipe for reading */
    rdfd = open(NP2, O_RDONLY);

    /* Write to the pipe */
    write(wrfd, argv[1], strlen(argv[1]));

    /* Read from the pipe */
    numread = read(rdfd, rdbuf, MAX_BUF_SIZE);

    rdbuf[numread] = '0';

    printf("Full Duplex Client : Read From the
    Pipe : %sn", rdbuf);
}
```

Running the Client and the Server

When you run the server, it will create the two named pipes and will block on the read call. It will wait until the client writes something to the named pipe. After that it will convert the string to upper case and then write it to the other pipe, which will be read by the client and displayed on STDOUT. When you run the client you will need to give a string as an argument.

Make sure you run the server first, so that the named pipe gets created.

Expected output:

1. Run the server:

   ```
   % fd_server &
   ```

   The server program will block here, and the shell will return control to the command line.

2. Run the client:

```
% fd_client hello
```

   The client program will send the string to server and block on the read to await the server's response.

3. The server prints the following:

```
Full Duplex Server : Read From the pipe : hello
```

The client prints the following:

```
Full Duplex Client : Read From the pipe : HELLO
```