Chapter 8 Interprocess Communication, Part I

Concepts Covered

Pipes
I/O Redirection
FIFOs
Concurrent Servers
Daemons
Multiplexed I/O with select()
API: dup, dup2, fpathconf, mkfifo, mknod, pipe, pclose, popen, select, setsid, shutdown, syslog, tee.

8.1 Introduction

Processes that cooperate in order to complete one or more tasks almost always need to communicate with each other. Sometimes the communication requires sharing data. One method of sharing data is by sharing a common file. If at least one of the processes modifies the file, then the file must be accessed in mutual exclusion. Sharing a file is essentially like sharing a memory-resident resource in that both are a form of communication that uses a shared resource that is accessed in mutual exclusion. Another paradigm involves passing data back and forth through some type of communication channel that provides the required mutual exclusion. A pipe is an example of this, as is a socket. This type of communication is broadly known as a *message-passing* solution to the problem.

This chapter is concerned only with message-passing types of communication. We will begin with *unnamed pipes*, after which we will look at *named pipes*, also known as *FIFO*'s, and then look at *sockets*. Part I is exclusively related to pipes.

8.2 Unnamed Pipes

You are familiar with how to use pipes at the command level. A command such as

```
$ last | grep 'reboot'
```

connects the output of `last` to the input of `grep`, so that the only lines of output will be those lines of `last` that contain the word '`reboot`'. The '`|`' is a `bash` operator; it causes `bash` to start the `last` command and the `grep` command *simultaneously*, and to direct the standard output of `last` into the standard input of `grep`.

Although '`|`' is a `bash` operator, it uses the lower-level, underlying *pipe* facility of UNIX, which was invented by Douglas Mcilroy, and was incorporated into UNIX in 1973. You can visualize the pipe mechanism as a special file or buffer that acts quite literally like a physical pipe, connecting the output of `last` to the input of `grep`, as in Figure 8.1.

The `last` program does not know that it is writing to a pipe and `grep` does not know that it is reading from a pipe. Moreover, if `last` tries to write to the pipe faster than `grep` can drain it, `last`

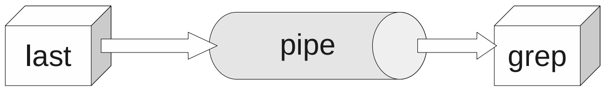

Figure 8.1: A pipe connecting `last` to `grep`.

will block, and if `grep` tries to read from an empty pipe because it is reading faster than `last` can write, `grep` will block, and both of these actions are handled behind the scenes by the kernel.

What then is a pipe? Although a pipe may seem like a file, it is not a file, and there is no file pointer associated with it. It is conceptually like a conveyor belt consisting of a fixed number of logical blocks that can be filled and emptied. Each write to the pipe fills as many blocks as are needed to satisfy it, provided that it does not exceed the maximum pipe size, and if the pipe size limit was not reached, a new block is made available for the next write. Filled blocks are conveyed to the read-end of the pipe, where they are emptied when they are read. These types of pipes are called `unnamed pipes` because they do not exist anywhere in the file system. They have no names.

An unnamed pipe[1] in UNIX is created with the `pipe()` system call.

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

The system call `pipe(fd)`, given an integer array `fd` of size 2, creates a pair of file descriptors, `fd[0]` and `fd[1]`, pointing to the "read-end" and "write-end" of a pipe inode respectively. If it is successful, it returns a 0, otherwise it returns -1. The process can then write to the write-end, `fd[1]`, using the `write()` system call, and can read from the read-end, `fd[0]`, using the `read()` system call. The read and write-ends are opened automatically as a result of the `pipe()` call. Written data are read in *first-in-first-out* (*FIFO*) order. The following program (`pipedemo0.c` in the demos directory) demonstrates this simple case.

Listing 8.1: pipedemo0.c

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

#define  READ_END   0
#define  WRITE_END  1
#define  NUM        5
#define  BUFSIZE   32

int main(int argc, char* argv[] )
{
    int    i, nbytes;
    int    fd[2];
    char   message[BUFSIZE+1];
```

[1]Unless stated otherwise, the word "pipe" will always refer to an unnamed pipe.

```
    if ( -1 == pipe(fd)) {
        perror("pipe call");
        exit(2);
    }

    for ( i = 1; i <= NUM; i++ ) {
        sprintf(message, "hello #%2d\n", i);
        write(fd[WRITE_END], message, strlen(message));
    }
    close(fd[WRITE_END]);

    printf("%d messages sent; sleeping a bit. Please wait...\n", NUM);
    sleep(3);

    while ( ( nbytes = read( fd[READ_END], message, BUFSIZE)) != 0 )
    {
        if ( nbytes > 0 ) {
            message[nbytes] = '\0';
            printf("%s", message);
        }
        else
            exit(1);
    }
    fflush(stdout);
    exit(0);
}
```

Notes.

- In this program, the write calls are not error-checked, which they should be. The `read()` in the while loop condition is error-checked: if it returns something strictly less than zero, `exit(1)` is executed.
- The `read()` call is a blocking read by default; you have to explicit make it non-blocking if you want it to be so. By design, a blocking read on a pipe will block waiting for data as long as the write-end of the pipe is held open. If the program does not close the write-end of the pipe before the read-loop starts, it will hang forever, because `read()` will continue to wait for data. This could be avoided if the read-loop knew in advance exactly how many bytes to expect, because in that case it could just read exactly that many bytes and then exit the loop, but it is rarely the case that one knows how much data to expect. Naturally, the process has to write the data into the pipe *before* the read loop begins, otherwise there will be nothing to read!
- Notice that the `read()` calls always read the same amount of data. This example demonstrates that the reader can read fixed-size chunks and assemble them into larger chunks, because *the data arrives in the order it was sent* (unlike data sent across a network.) Pipes have no concept of message boundaries – they are simply byte streams.
- Finally, observe that before calling `printf()` to print the string on the standard output, the string has to be null-terminated.

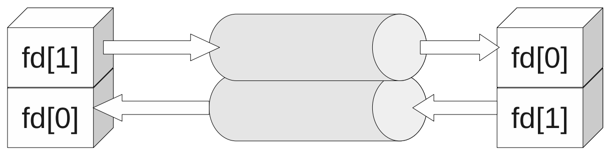

Figure 8.2: Parent and child sharing a pipe.

The semantics of reading from a pipe are much more complex than reading from a file. The following table summarizes what happens when a process tries to read *n* bytes from a pipe that currently has *p* bytes in it that have not yet been read.

Pipe Size (p)	**At least one process has the pipe open for writing**			**No processes have the pipe open for writing**
	Blocking read		**Non-blocking read**	
	At least one writer is sleeping	**No writer is sleeping**		
p = 0	Copy n bytes and return n, waiting for data as necessary when the pipe is empty.	Block until data is available, copy it and return its size.	Return `-EAGAIN`.	Return 0.
0 < p < n		Copy p bytes and return p, leaving the buffer empty.		
p ≥ n	Copy n bytes and return n leaving p-n bytes in the pipe buffer.			

The semantics depend upon whether or not a writer has been put to sleep because it tried to write into the pipe previously but the pipe was full. On a non-blocking read request, if the number of bytes requested, *n*, is greater than what is currently in the pipe and at least one writer is in this sleeping state, then the read will attempt to read *n* bytes, because as he pipe is emptied, the writer will be awakened to write into the pipe. If no writer is sleeping and the pipe is empty, however, then the read will block until some data becomes available.

8.2.1 Parent and Child Sharing a Pipe

Of course there is little reason for a process to create a pipe to write messages to itself. Pipes exist in order to allow two different processes to communicate. Typically, a process will create a pipe, and then fork a child process. After the fork, the parent and child will each have copies of the read and write-ends of the pipe, so there will be two data channels and a total of four descriptors, as shown in Figure 8.2.

On some Unix systems, such as System V Release 4 Unix, pipes are implemented in this full-duplex mode, allowing both descriptors to be written into and read from at the same time. POSIX allows only *half-duplex mode*, which means that data can flow in only one direction through the pipe, and each process must close one end of the pipe. The following illustration depicts this half-duplex mode.

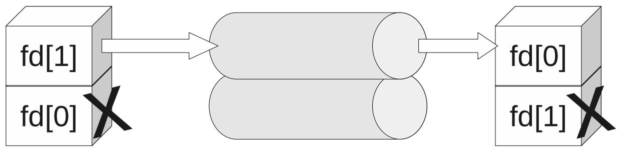

Figure 8.3: Pipe in half-duplex mode.

The paradigm for half-duplex use of a pipe by two processes is as follows:

```
if ( -1 == pipe(fd))
    exit(2);  // failed to create pipe
switch ( fork() ) {
    // child process:
case 0:
    close(fd[1]);  // close write-end
    bytesread = read( fd[0], message, BUFSIZ);
    // check for errors afterward of course
    break;
// parent process:
default:
    close(fd[0]);  // close read-end
    byteswritten = write(fd[1], buffer, strlen(buffer) );
    // and so on
    break;
}
```

Linux follows the POSIX model but does not require each process to close the end of the pipe it is not going to use. However, for code to be portable, it should follow the POSIX model. All examples here will assume half-duplex mode. The following is the first example of two-process communication through a pipe. The parent process reads the command line arguments and sends them to the child process, which prints them on the screen. As we get more deeply involved with pipes, you will discover that it is easy to make mistakes when coding for them, as there are many intricacies to be aware of. This first program exposes a few of them.

Listing 8.2: pipedemo1.c

```
#include <stdio.h>
#include <string.h>
#include <unistd.h>
#include <stdlib.h>
#include <sys/wait.h>

#define  READ_FD   0
#define  WRITE_FD  1

int main(int argc, char* argv[] )
```

```
{
    int i;
    int bytesread;
    int fd[2];
    char  message[BUFSIZ];

    /* check proper usage */
    if ( argc < 2 ) {
        fprintf(stderr, "Usage:  %s message\n", argv[0]);
        exit(1);
    }

    /* try to create pipe */
    if ( -1 == pipe(fd)) {
        perror("pipe call");
        exit(2);
    }

    /* create child process */
    switch ( fork() ) {
    case -1:
        /* fork failed -- exit */
        perror("fork()");
        exit(3);

    case 0: /* child code */
        /* Close write end, otherwise child will never terminate */
        close(fd[WRITE_FD]);
        /* Loop while not end of file or not a read error    */
        while ( ( bytesread = read( fd[READ_FD], message, BUFSIZ) )
                != 0 )
            if ( bytesread > 0 ) {  /* more data */
                message[bytesread] = '\0';
                printf("Child received the word: '%s'\n", message);
                fflush(stdout);
            }
            else {  /*read error */
                perror("read()");
                exit(4);
            }
        exit(0);

    default: /* parent code */
        close(fd[READ_FD]);   /* Close read end, since parent is writing */
        for ( i = 1; i < argc; i++ )
            /* send each word separately */
            if ( write(fd[WRITE_FD], argv[i], strlen(argv[i]) ) != -1 )
            {
                printf("Parent sent the word: '%s'\n", argv[i]);
                fflush(stdout);
            }
            else {
                perror("write()");
                exit(5);
            }
        close(fd[WRITE_FD]);

        /* wait for child so it does not remain a zombie */
        /* don't care about it's status, so pass a NULL pointer */
        if (wait(NULL) == -1) {
            perror("wait failed");
            exit(2);
        }
    }
    exit(0);
}
```

Notes.

- It is now critical that the child closes the write-end of its pipe before it starts to read. As was noted earlier, reads are blocking by default and will remain waiting for input as long as ANY write-end of the pipe is open, including its own. Therefore, not only do we want to close the unused end of the pipe for the code to be more portable, but also for it to be correct!
- The parent waits for the child process because if it does not, the child will become a zombie in the system. You should make a habit of waiting for all processes that you create.
- The output of the parent and child on the terminal may occur in any order. This program makes no attempt to coordinate the use of the terminal simply because it would distract from its purpose as a demonstration of how to use pipes.

8.2.2 Atomic Writes

In a POSIX-compliant system, a single write will be executed atomically as long as the number of bytes to be written does not exceed `PIPE_BUF`. This means that if several processes are each writing to the pipe at the same time, as long as each limits the size of each write to N $\leq$ `PIPE_BUF` bytes, the data will not be intermingled. If there is not enough room in the pipe to store N $\leq$ `PIPE_BUF` bytes, and writes are blocking (the default), then `write()` will be blocked until room is available. On the other hand, if N $>$ `PIPE_BUF`, there is no guarantee that the writes will be atomic.

To use the value of `PIPE_BUF` in a program, include the header file `<limits.h>`. For example,

```
    #include <limits.h>
    char chunk[PIPE_BUF];
```

In the event that your `<limits.h>` header file does not define `PIPE_BUF`, it means that the value is greater than the POSIX minimum value for this constant, which is `POSIX_PIPE_BUF`, and is usually 512 bytes. POSIX does not require that `PIPE_BUF` be defined in this case. Therefore, you should write the above code snippet as

```
    #include <limits.h>
    #ifndef PIPE_BUF
```

```
        #define PIPE_BUF  POSIX_PIPE_BUF;
    #endif

    char chunk[PIPE_BUF];
```

An alternative that may work on your system is to use the `fpathconf()` system call to determine the value of the atomic write size dynamically. The `fpathconf()` system call returns the value of various system dependent configuration values associated with an open file descriptor. The `fpathconf()` function's synopsis and description is

```
    #include <unistd.h>

    long fpathconf(int filedes, int name);
    long pathconf(char *path, int name);

    DESCRIPTION
    fpathconf() gets a value for the configuration option name
    for the open file descriptor filedes.
```

The second argument to `fpathconf()` is a mnemonic name defined in the man page. These are names such as `_PC_NAME_MAX`, `_PC_PATH_MAX`, and `_PC_PIPE_BUF`. Each name has a different usage, and its validity depends upon whether the given file descriptor is that of a file, a directory, a pipe, or a terminal. If `filedes` is a pipe, then the constant `_PC_PIPE_BUF` *is supposed to tell* `fpathconf()` to return the maximum number of bytes that may be written atomically to that pipe. It may not return this value. This will be explained below.

Although it is not necessary to know the value of `PIPE_BUF`, it is an interesting exercise to learn its value and make sure that it has the magical properties it is supposed to have. The following program is designed to demonstrate (but not prove) that writes of up to `PIPE_BUF` bytes are atomic, and that larger writes may not be atomic. It also demonstrates how to create multiple writers and a single reader. The program creates two writer processes and one reader. One writer writes '`X`'s into the pipe, the other '`y`'s. They each write the same number of characters each time. The command line argument specifies the number of writes that each makes to the pipe. The idea is that if the number is large enough the scheduler will time slice them often enough so that one will write for a while, then the next, and so on. The parent is the reader. It reads the data from the pipe and stores it in a file. The parent reads smaller chunks since it does not matter.

Listing 8.3: pipedemo2.c

```
#include <stdio.h>
#include <string.h>
#include <unistd.h>
#include <stdlib.h>
#include <fcntl.h>
#include <limits.h>
#include <signal.h>
#include <sys/wait.h>
```

```
#define  READ_FD   0
#define  WRITE_FD  1
#define  RD_CHUNK     10
#define  ATOMIC

#ifndef PIPE_BUF
    #define PIPE_BUF POSIX_PIPE_BUF
#endif

void do_nothing( int signo)
{
  return;
}


int main(int argc, char* argv[] )
{
    int i, repeat;
    int bytesread;
    int mssglen;
    pid_t child1, child2;
    int fd[2];
    int outfd;
    char  message[RD_CHUNK+1];
    char  *Child1_Chunk, *Child2_Chunk;
    long Chunk_Size;

    static struct sigaction sigact;

    sigact.sa_handler = do_nothing;
    sigfillset(&(sigact.sa_mask));
    sigaction(SIGUSR1, &sigact, NULL);


    /* check proper usage */
    if ( argc < 2 ) {
        fprintf(stderr, "Usage:  %s size\n", argv[0]);
        exit(1);
    }

    /* try to create pipe */
    if ( -1 == pipe(fd)) {
        perror("pipe call");
        exit(2);
    }

    repeat     = atoi(argv[1]);
#if defined ATOMIC
    Chunk_Size = PIPE_BUF;
#else
    Chunk_Size = PIPE_BUF + 200;
#endif
    printf("Chunk size = %ld\n", Chunk_Size);
    printf("Value of PIPE_BUF is %d\n", PIPE_BUF);
    Child1_Chunk =  calloc(Chunk_Size, sizeof(char));
    Child2_Chunk =  calloc(Chunk_Size, sizeof(char));
    if ( ( NULL == Child1_Chunk  ) ||
         ( NULL == Child2_Chunk  ) ) {
        perror("calloc");
        exit(2);
    }

    /* create the string that child1 writes */
    Child1_Chunk[0] = '\0'; /* just to be safe */
    for ( i = 0; i < Chunk_Size-2; i++)
        strcat(Child1_Chunk, "X");
    strcat(Child1_Chunk,"\n");

    /* create the string that child2 writes */
    Child2_Chunk[0] = '\0'; /* just to be safe */
    for ( i = 0; i < Chunk_Size-2; i++)
        strcat(Child2_Chunk, "y");
    strcat(Child2_Chunk,"\n");

    /* create first child process */
    switch ( child1 = fork() ) {
    case -1:  /* fork failed -- exit */
        perror("fork()");
        exit(3);

    case 0:   /* child1 code */
        mssglen = strlen(Child1_Chunk);
        pause();
        for ( i = 0; i < repeat; i++ ) {
            if ( write(fd[WRITE_FD], Child1_Chunk, mssglen )
                 != mssglen ) {
                perror("write");
                exit(4);
            }

        }
        close(fd[WRITE_FD]);
        exit(0);
    default:  /* parent creates second child process */
        switch ( child2 = fork() ) {
        case -1:  /* fork failed -- exit */
            perror("fork()");
            exit(5);

        case 0:   /* child2 code */
            mssglen = strlen(Child2_Chunk);
            pause();
            for ( i = 0; i < repeat; i++ ) {
                if ( write(fd[WRITE_FD], Child2_Chunk, mssglen )
                     != mssglen ) {
                    perror("write");
                    exit(6);
                }

            }
            close(fd[WRITE_FD]);
            exit(0);
        default:  /* parent code */
            outfd = open("pd2_output",
                         O_WRONLY | O_CREAT | O_TRUNC, 0644);
            if ( -1 == outfd ) {
                perror("open");
                exit(7);
            }

            close(fd[WRITE_FD]);
            kill(child1, SIGUSR1);
            kill(child2, SIGUSR1);
            while ( ( bytesread = read( fd[READ_FD], message, RD_CHUNK) )
                       != 0 )
                if ( bytesread > 0 ) {   /* more data */
                    write(outfd, message, bytesread);
                }
                else {                   /*read error */
                    perror("read()");
                    exit(8);
                }

            close(outfd);
            /* collect zombies */
            for ( i = 1; i <= 2; i++ )
                if ( wait(NULL) == -1) {
                    perror("wait failed");
                    exit(9);
                }
            close(fd[READ_FD]);
            free(Child1_Chunk);
            free(Child2_Chunk);
        }
        exit(0);
    }
}
```

Notes.

- The parent process is the reader; the two child processes are writers. Each child calls `pause()` to start so that neither gets to grab the processor immediately. The parent sends a `SIGUSR1` signal to them when it is ready to start reading from the pipe.
- Each child write a chunk of size `Chunk_Size` into the pipe. `Chunk_Size` is either `PIPE_BUF` or 200 bytes larger than it.
- The parent reads from the pipe and writes the data into a file named `pd2_output`. When the `read()` returns 0, the children have finished writing and closed the pipe, so the parent closes the output file and calls `wait()` to collect the exit status of the children.

- The program prints the value of PIPE_BUF and the actual chunk size before the pipe operations begin.

First compile the program as it is written, naming the executable pipedemo2. When pipedemo2 is run, the output will show that writes are atomic – each string written by each child in a single write has a newline at its end, and in the output, every sequence of X's will be terminated by a newline and every sequence of y's will end in a newline. There will be no occurrence of the string Xy or yX in the output because the kernel serializes the concurrent writes, and each time a child process writes, it writes its entire string, either X's or y's. The output does not prove it is atomic; it just shows that no output was intermingled, and thus no write was interrupted.

Each child should write two thousand times or more in order for us to see the possibility of their each competing for the shared pipe, so the program should be run with a command line argument of 2000 or more. It would be tedious to check the output by hand to determine whether there are any lines with intermingled output. The following script is designed to do this automatically:

```
#!/bin/bash
if [[ $# < 1 ]]
then
    printf "Usage: %b repeats\n" $0
    exit
fi

pipedemo2  $1
printf "Number of X lines     : "
    grep X pd2_output | wc -l
printf "Number of y lines     : "
    grep y pd2_output | wc -l
printf "X lines in first %b : " $1
    head -$1 pd2_output | grep X | wc -l
printf "y lines in first %b : " $1
    head -$1 pd2_output | grep y | wc -l
printf "X lines in last %b  : " $1
    tail -$1 pd2_output | grep X | wc -l
printf "y lines in last %b  : " $1
    tail -$1 pd2_output | grep y | wc -l
printf "Xy lines              : "
    grep Xy pd2_output  | wc -l
printf "yX lines              : "
    grep yX pd2_output  | wc -l
```

The command line argument is the number of chunks that each child should write. The script summarizes the output. If repeats is 1000, You should see output something like

```
Number of X lines : 1000
Number of y lines : 1000
X lines in first 1000 : 515
```

```
y lines in first 1000 : 485
X lines in last 1000 : 485
y lines in last 1000 : 515
Xy lines : 0
yX lines : 0
```

The last two lines show that all writes were atomic because there are no lines that contain an `Xy` or `yX` combination. Now edit the program by commenting out the line

```
#define ATOMIC
```

and recompile it. This flag determines how large the chunk is. When it is turned off, the chunk is larger than `PIPE_BUF` bytes. Run the script again. The output will most likely look something like this:

```
Number of X lines : 1443
Number of y lines : 1437
X lines in first 1000 : 758
y lines in first 1000 : 718
X lines in last 1000 : 685
y lines in last 1000 : 719
Xy lines : 577
yX lines : 586
```

which shows that when the chunk size of a write exceeds `PIPE_BUF`, the writes will not be atomic.

8.2.2.1 More About fpathconf()

Almost all systems comply with the POSIX requirement that result of the call

```
pipe_size = fpathconf(fd, _PC_PIPE_BUF);
```

is the system's current value of `PIPE_BUF`. But not all do. Some systems using recent versions of the GNU C Library will use a different version of `fpathconf()`. This version returns the pipe capacity, not the value of `PIPE_BUF`, but only if the kernel supports it. Linux kernels after 2.6.35 do for certain. What this implies is that you cannot reliably use the result of `fpathconf()` to determine the maximum number of bytes in an atomic write on all systems.

8.2.3 Pipe Capacity

The capacity of a pipe may be larger than `PIPE_BUF`. There is no exposed system constant that indicates the total capacity of a pipe; however, the following program, based on one from [Haviland *et al*], can be run on any system to test the maximum capacity of a pipe, and also to prove that a process cannot write to a pipe unless it has at least `PIPE_BUF` bytes available.

Listing 8.4: pipesizetest.c

```
#include <stdio.h>
#include <string.h>
#include <unistd.h>
#include <stdlib.h>
#include <signal.h>
#include <limits.h>

int          count = 0;
sig_atomic_t full  = 0;

/**
 *  The SIGALRM handler. This sets the full flag to indicate that the
 *  write call blocked, and it prints the number of characters written
 *  to the pipe so far.
 */
void on_alarm( int signo)
{
    printf("\nwrite() blocked with %d chars in the pipe.\n", count);
    full = 1;
}

int main(int argc, char* argv[])
{
    int fd[2];
    int pipe_size;
    int bytesread;
    int amount_to_remove;

    char buffer[PIPE_BUF];
    char c = 'x';
    static struct sigaction sigact;

    sigact.sa_handler = on_alarm;
    sigfillset(&(sigact.sa_mask));
    sigaction(SIGALRM, &sigact, NULL);

    if ( -1 == pipe(fd) ) {
        perror("pipe failed");
        exit(1);
    }

    /* Check whether the _PC_PIPE_BUF constant returns the pipe capacity
       or the atomic write size
    */
    pipe_size = fpathconf(fd[0], _PC_PIPE_BUF);

    while (1) {
        /* Set an alarm long enough that if write fails it will fail  */
        /* within this amount of time. 8 seconds is long enough. */
        alarm(4);
        write(fd[1], &c, 1);
        /* Unset the alarm */
        alarm(0);
        /* Did alarm expire? If so, write failed and we stop the loop */
        if ( full )
            break;

        /* Report how many chars written so far */
        if ( (++count % 1024) == 0 )
            printf ( "%d chars in pipe\n", count);
    }

    printf( "The maximum number of bytes that the pipe stored is %d.\n",
            count);
    printf( "The value returned by fpathconf(fd,_PC_PIPE_BUF) is %d.\n\n",
            pipe_size);

    printf( "Now we remove characters from the pipe and demonstrate that"
            " we cannot\n"
            "write into the pipe unless it has %d (PIPE_BUF) free bytes.\n",
             PIPE_BUF);

    amount_to_remove = PIPE_BUF-1;

    printf( "First we remove %d characters (PIPE_BUF-1) and try to "
            "write into the pipe.\n", amount_to_remove);

    full = 0;
    bytesread = read(fd[0], &buffer, amount_to_remove);
    if ( bytesread < 0 ) {
        perror("error reading pipe");
        exit(1);
    }
    count = count - bytesread;
    alarm(4);
    write(fd[1], &c, 1);
    /* Unset the alarm */
    alarm(0);
    if ( full )
        printf( "We could not write into the pipe.\n");
    else
        printf( "We successfully wrote into the pipe.\n");

    amount_to_remove = PIPE_BUF - amount_to_remove;
    full = 0;

    printf( "\nNow we remove one more character and try to "
            "write into the pipe.\n");

    bytesread = read(fd[0], &buffer, amount_to_remove );
    if ( bytesread < 0 ) {
        perror("error reading pipe");
        exit(1);
    }
    count = count - bytesread;
    alarm(4);
    write(fd[1], &c, 1);
    /* Unset the alarm */
    alarm(0);
    if ( full )
        printf( "We could not write into the pipe.\n");
    else
        printf( "We successfully wrote into the pipe.\n");

    return 0;
}
```

Notes.

- The main program is the only process, and within its loop it repeatedly writes a single character to the pipe.
- Since the program never reads from the pipe, the pipe will eventually fill up. When the process attempts to write to the pipe after it is full, it will be blocked. To prevent it from being blocked forever, it sets an alarm before each `write()` call and unsets it afterwards. The alarm interval, 10 seconds, is long enough so that the alarm will never expire before a successful write finishes. When the pipe is full however, the write will be blocked indefinitely, and therefore the alarm will expire, interrupting the `write()`, and the alarm handler will display the total number of bytes written so far and then terminate the program.
- The program reports the value of `fpathconf(fd,_PC_PIPE_BUF)` in order to compare it to the actual pipe capacity. On Linux systems using recent versions of the GNU C Library, this value will be the pipe capacity, not the current value of `PIPE_BUF`.
- Once the pipe is full, the program removes `PIPE_BUF`-1 bytes from the pipe and attempts to write to it. This will fail. It then removes one more byte so that the pipe has `PIPE_BUF` bytes free, and writes to it again. This time the write will succeed.
- The program displays messages to indicate the various successes and failures.

8.2.4 Caveats and Reminders Regarding Blocking I/O and Pipes

Quite a bit can go wrong when working with pipes. These are some important facts to remember about using pipes and non-blocking reads and writes. Some of these have been mentioned already, some not. This section consolidates them into a single place.

1. If a `write()` is made to a pipe that is not open for reading by any process, a `SIGPIPE` signal will be sent to the writing process, which, if not caught, will terminate that process. If it is caught, after the `SIGPIPE` handler finishes, the `write()` will return with a -1, and `errno` will be set to the value `EPIPE`.
2. If there are one or more processes writing to a pipe, if a reading process closes its read-end of the pipe and no other processes have the pipe open for reading, each writer will be sent the `SIGPIPE` signal, and the same rules mentioned above regarding handling of the signal apply to each process.

3. As long as one writer has a pipe open for writing, a call to `read()` will remain blocked until there is data in the pipe. Therefore, if all writers finish writing to the pipe, but a single writer fails to close the write-end of the pipe, if a reader calls `read()`, the reader will remain permanently blocked. Once all writers close the write-ends of the pipe, the `read()` will return zero.
4. A `write()` to a full pipe will block the writer until there are `PIPE_BUF` free bytes in the pipe.
5. Unlike reads from a file, `read()` requests to a pipe drain the pipe of the data that was read. Therefore, when multiple readers read from the same pipe, no two read the same data.
6. Writes are atomic as long as the number of bytes is smaller than `PIPE_BUF`.
7. Reads are atomic in the sense that, if there is any data in the pipe when the call is initiated, the `read()` will return with as much data as is available, up to the number of bytes requested, and it is guaranteed not to be interrupted.
8. Processes cannot `seek()` on a pipe.

The situation is entirely different with non-blocking reading and writing. These will be discussed later. However, before continuing with the discussion of pipes, we will take a slight detour to look at I/O redirection in general, because studying I/O redirection will give us insight into some of the ways in which pipes are used.

8.3 I/O Redirection Revisited

8.3.1 Simulating Output Redirection

How does the shell implement I/O redirection? The key to understanding this rests on one simple principle used by the kernel: the `open()` system call always chooses the lowest numbered available file descriptor.

Suppose that you have entered the command

```
$ ls > listing
```

The steps taken by the shell are

1. `fork()` a new process.
2. In the new process, `close()` file descriptor 1 (standard output).
3. In the new process, `open()` (with the `O_CREAT` flag) the file named `listing`.
4. Let the new process `exec()` the `ls` program.

After step 1, the child and parent each have copies of the same file descriptors. After step 2, the child has closed standard output, so file descriptor 1 is free. In step 3, the kernel sees descriptor 3 is free, so it uses descriptor 3 to point to the file structure for the file named `listing`. Then the child calls `exec()` passing it the string `"ls"`. The `ls` program writes to file descriptor 1, usually standard output, but in fact it is really writing to the file named `listing`. In the meanwhile, the shell continues to have descriptor 1 pointing to the standard output device, so it is unaffected by this secret trick it played on the `ls` command.

The following program, called `redirectout.c`, illustrates how this works. It simulates the shell's '`>`' operator. It forks a child, closes standard output descriptor 1, opens the output file specified in `argv[2]` for writing, and execs `argv[1]`. The parent simply waits for the child to terminate. Compile it and name it `redirectout`, and then try a command such as the following:

```
$ redirectout who whosloggedon
```

Redirecting standard input works similarly. The only difference is that the process has to close the standard input descriptor 0, and then open a file for reading.

Listing 8.5: redirectout.c

```
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <fcntl.h>
#include <sys/wait.h>

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

    if ( argc < 3 ) {
        fprintf(stderr, "Usage: %s  command output-file\n", argv[0]);
        exit(1);
    }

    switch ( fork()) {
    case -1:
        perror("fork");
        exit(1);
    case  0:   /* child code */
        /* Close standard output */
        close(1);
        /* Open the file into which to redirect standard output */
        /* and check that it succeeds */
        if ( (fd = open( argv[2], O_WRONLY | O_CREAT | O_TRUNC, 0644 ))
            == -1 )
             exit(1);

        /* execute the command in argv[1] */
        execlp( argv[1],argv[1], NULL );

        /* should not reach here!     */
        perror("execlp");
        exit(1);
    default:   /* parent code; just waits for child */
        wait(NULL);
    }
    return 0;
}
```

8.3.2 Simulating the '|' Shell Operator

The pertinent question now is, how can we write a similar program that can simulate how the shell carries out a command such as

```
$ last | grep 'pts/2'
```

This cannot be accomplished using just the `open()`, `close()`, and `pipe()` system calls, because we need to connect one end of a pipe to the standard output for `last`, and the other end to the standard input for `grep`. Just closing file descriptors cannot do this. There are two system calls that can be used for this purpose: `dup()` and `dup2()`. `dup()` is the progenitor of `dup2()`, which superseded it. We will first look at a solution using `dup()`.

The `dup()` system call duplicates a file descriptor. From the man page:

```
#include <unistd.h>
int dup(int oldfd);

After a successful return from dup(), the old and new file descriptors
may be used interchangeably. They refer to the same open file description
(see open(2)) and thus share file offset and file status flags;
for example, if the file offset is modified by using lseek(2) on one
of the descriptors, the offset is also changed for the other.
```

In other words, given a file descriptor, `oldfd`, `dup()` creates a new file descriptor that points to the same kernel file structure as the old one. But again the critical feature of `dup()` is that it returns the lowest-numbered available file descriptor. Therefore, consider the following sequence of actions.

```
int fd[2];                  /* Declare descriptors for a pipe  */
pipe(fd);                   /*  Create the pipe */
switch ( fork() )           /* Fork a child */
case 0:                     /* In the child: */
    close(fileno(stdout));      /* close standard output  */
    dup(fd[1]);                 /* dup write-end of pipe  */
    close(fd[0]);               /* close read-end of pipe */
    exec("last", "last", NULL); /* exec the command that writes to the pipe */
```

The `dup()` call will find the standard output file descriptor available, and since that is the lowest numbered available descriptor, it will make that point to the same structure as `fd[1]` points to. Therefore, when the `last` command writes to standard output, it will really be writing to the write-end of the pipe.

Now it is not hard to imagine what the parent's job is. It has to close the standard input descriptor, then `dup(fd[0])`, and exec the `grep` command. We can put these ideas together in a more general program, called `shpipe1.c`, which follows.

Listing 8.6: shpipe1.c

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

int main(int argc, char* argv[])
{
    int  fd[2];

    if ( argc < 3 ) {
        fprintf(stderr, "Usage: %s  command1 command2\n", argv[0]);
        exit(1);
    }

    if ( -1 == pipe(fd)) {
        perror("pipe call");
        exit(2);
    }

    switch ( fork()) {
    case -1:
        perror("fork");
        exit(1);
    case  0:
        close(fileno(stdout));
        dup(fd[1]);
        close(fd[0]);   /* close read end since child does not use it */
        close(fd[1]);   /* close write end since it is not needed now */
        execlp( argv[1],argv[1], NULL );
        perror("execlp");
        exit(1);
    default:
        close(fileno(stdin));
        dup(fd[0]);
        close(fd[1]);  /* close write end to prevent child from blocking */
        close(fd[0]);  /* close read end since it is not needed now */
        execlp( argv[2],argv[2], NULL );
        exit(2);
  }
    return 0;
}
```

If you compile this and name it `shpipe1`, then you can try commands such as

```
$ shpipe1 last more
```

and

```
$ shpipe1 ls wc
```

There is a problem here. For one, the parent cannot wait for the child because it uses `execlp()` to replace its image. This can be solved by forking two children and letting the second do the work of the reading process. More importantly, this solution is not general, because there are two steps – close standard output and then `dup()` the write end of the pipe. There is a small window of time between closing standard output and duplicating the write-end of the pipe in which the child could be interrupted by a signal whose handler might close file descriptors so that the descriptor returned by `dup()` will not be the one that was just closed.

This is the reason that `dup2()` was created. `dup2(fd1, fd2)` will duplicate `fd1` in `fd2`, closing `fd2` if necessary, as a single atomic operation. In other words, if `fd2` is open, it will close it, and make `fd2` point to the same file structure to which `fd1` pointed. Its man page entry is

```
    #include <unistd.h>
    int dup2(int oldfd, int newfd);
    .....
    dup2() makes newfd be the copy of oldfd, closing newfd first if necessary.
```

(`dup()` and `dup2()` share the same page. I deleted `dup2()`'s description above. This is the relevant part of it.)

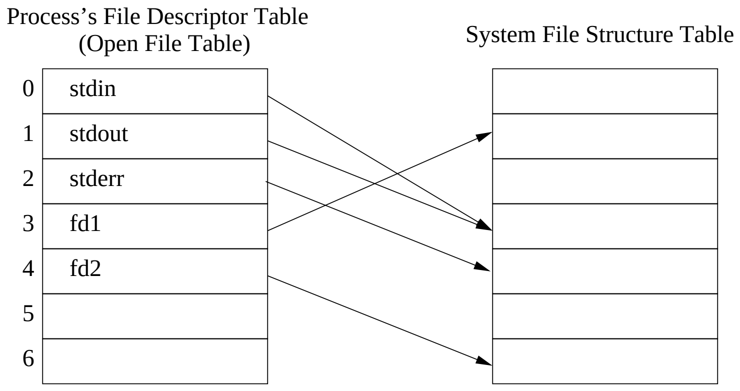

Figure 8.4: Initial state of open file table.

A picture best illustrates how `dup2()` works. Assume the initial state of the file descriptors for the process is as shown in Figure 8.4. Now suppose that the process makes the call

```
    dup2( fd2 , fileno(stdin));
```

Then, after the call the table is as shown in Figure 8.5. Descriptor 0 (standard input) became a copy of `fd2` as a result of the call. Descriptor `fd2` is now redundant and can be closed if `stdin` is going to be used instead.

The following listing is of a program, `shpipe2.c` , which is an improved version of `shpipe1.c`. that uses the `dup2()` call instead of `dup()`.

Listing 8.7: shpipe2.c

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

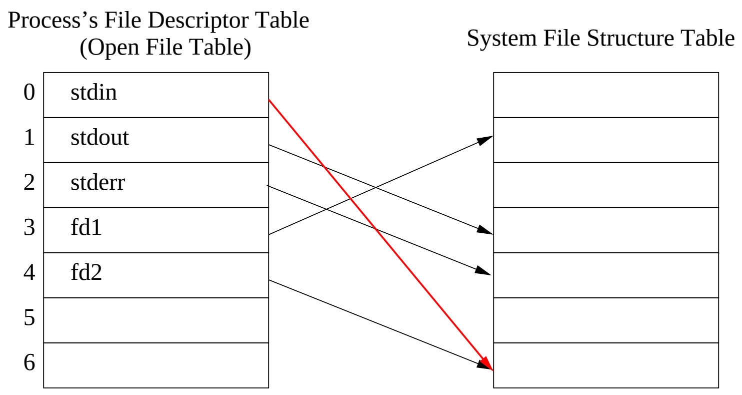

Figure 8.5: State of open file table after dup2(fd2,fileno(stdin));

```
#include <stdlib.h>
#include <fcntl.h>
#include <sys/wait.h>

int main(int argc, char* argv[])
{
    int     fd[2];
    int     i;
    pid_t   child1, child2;

    if ( argc < 3 ) {
        fprintf(stderr, "Usage: %s  command1 command2\n", argv[0]);
        exit(1);
    }

    if ( -1 == pipe(fd)) {
        perror("pipe call");
        exit(2);
    }

    switch ( child1 = fork()) {
    case -1:
        perror("fork");
        exit(1);
    case  0:        /* child1 */
        dup2(fd[1],fileno(stdout));         /* now stdout points to fd[1] */
        close(fd[0]);                       /* close input end of pipe */
        close(fd[1]);                       /* close output end of pipe */
        execlp( argv[1],argv[1], NULL );    /* run the first command    */
        perror("execlp");
        exit(1);
    default:
        switch ( child2 = fork()) {
        case -1:
            perror("fork");
            exit(1);
        case  0:        /* child2 */
            dup2(fd[0],fileno(stdin));      /* now stdin points to fd[0] */
            close(fd[0]);                  /* close input end of pipe */
            close(fd[1]);                  /* close output end of pipe */
            execlp(argv[2],argv[2], NULL ); /* run the first command */
            perror("execlp");
            exit(2);
        default:
            close(fd[0]);  /* parent must close its ends of the first pipe */
            close(fd[1]);
              for ( i = 1; i <= 2; i++ )
                 if ( wait(NULL) == -1) {
                      perror("wait failed");
                      exit(3);
                   }
            return 0;
        }
    }
    return 0;
}
```

There are a couple of things you can try to do at this point to test your understanding of pipes.

1. There is a UNIX utility called `tee` that copies its input stream to standard output as well as to its file argument:

   ```
   $ ls -l | tee listing
   ```

   will copy the output of "`ls -l`" into the file named listing as well as to standard output. Try to write your own version of `tee`.

2. Extend `shpipe2` to work with any number of commands so that

   ```
   $ shpipe3 cmmd cmmd ... cmmd
   ```

   will act like

   ```
   $ cmmd | cmmd | ... | cmmd
   ```

8.3.3 The popen() Library Function

The sequence

1. generate a pipe,
2. fork a child process,
3. duplicate file descriptors, and
4. execute a new program in order to redirect the input or output of that program to the parent,

is so common that the developers of the C library added a pair of functions, `popen()` and `pclose()` to streamline this procedure:

```
#include <stdio.h>
FILE *popen(const char *command, const char *type);
int pclose(FILE *stream);
```

The `popen()` function creates a pipe, forks a new process to execute the shell `/bin/sh` (which is system dependent), and passes the command to that shell to be executed by it (using the `-c` flag to the shell, which tells it to expect the command as an argument.)

`popen()` expects the second argument to be either `"r"` or `"w"`. If it is `"r"` then the process invoking it will be returned a `FILE` pointer to the *read-end* of the pipe and the write-end will be attached to the standard output of the command. If it is `"w"`, then the process invoking it will be returned a `FILE` pointer to the *write-end* of the pipe, and the read-end will be attached to the standard input of the command. The output stream is fully buffered.

File streams created with `popen()` must be closed with `pclose()`, which will wait for the invoked process to terminate and returns its exit status or -1 if `wait4()` failed.

An example will illustrate. We will write a third version of the `shpipe` program, called `shpipe3`, using `popen()` and `pclose()` instead of the `pipe()`, `fork()`, `dup()` sequence. See Listing 8.8 below.

Listing 8.8: shpipe3.c

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

int main(int argc, char* argv[])
{
    int     nbytes;
    FILE    *fin;                 /* read-end of pipe */
    FILE    *fout;                /* write-end of pipe */
    char    buffer[PIPE_BUF];     /* buffer for transferring data */

    if ( argc < 3 ) {
        fprintf(stderr, "Usage: %s  command1 command2\n", argv[0]);
        exit(1);
    }

    if ( (fin = popen(argv[1], "r")) == NULL ) {
        fprintf(stderr, "popen() failed\n");
        exit(1);
    }

    if ( (fout = popen(argv[2], "w")) == NULL ) {
        fprintf(stderr, "popen() failed\n");
        exit(1);
    }

    while ( (nbytes = read(fileno(fin), buffer, PIPE_BUF)) > 0 )
        write(fileno(fout), buffer, nbytes);

    pclose(fin);
    pclose(fout);
    return 0;
}
```

8.4 Named Pipes

Unnamed pipes are an elegant mechanism, however, they have several drawbacks. They can only be shared by processes with a common ancestor, such as a parent and child, or multiple children or descendants of a parent that created the pipe. Also, they cease to exist as soon as the processes that are using them terminate, so they must be recreated every time they are needed. If you are trying to write a server program with which clients can communicate, the clients will need to know the name of the pipe through which to communicate, but an unnamed pipe has no such name.

Named pipes make up for these shortcomings. A named pipe, or *FIFO*, is very much like an unnamed pipe in how you use it. You read from it and write to it in the same way. It behaves the same way with respect to the consequences of opening and closing it when various processes are either reading or writing or doing neither. In other words, the semantics of opening, closing, reading, and writing named and unnamed pipes are the same.

What distinguishes named pipes from unnamed pipes is that

- They exist as directory entries in the file system and therefore have associated permissions and ownership[2].
- They can be used by processes that are not related to each other.
- They can be created and deleted at the shell level or at the programming level.

8.4.1 Named Pipes at the Command Level

Before we look at how they are created within a program, let us look at how they are created at the user level. There are two commands to create a FIFO. The older command is `mknod`. `mknod` is a general purpose utility for creating device special files. There is also a `mkfifo` command, which can only be used for creating a FIFO file. We will look at how to use `mknod`. You can read about the `mkfifo` command in the man pages.

```
$ mknod PIPE p
```

creates a FIFO named `"PIPE"`. The lowercase `p` , which must follow the file name, indicates to `mknod` that `PIPE` should be a FIFO (`p` for pipe.) After typing this command, look at the working directory:

```
$ ls -l PIPE
prw-r--r-- 1 stewart stewart 0 Apr 30 22:29 PIPE|
```

[2]Although they have directory entries, they do not exist in the file system. They have no disk blocks and their data is not on disk when they are in use.

The 'p' file type indicates that PIPE is a FIFO. Notice that it has 0 bytes. Try the following command sequence:

```
$ cat < PIPE &
$ ls -l > PIPE; wait
```

If we do not put the cat command into the background it will hang because a process trying to read from a pipe will block until there is at least one process trying to write to it. The cat command is trying to read from PIPE and so it will not return and you will not get the shell prompt back without backgrounding it. The cat command will terminate as soon as it receives the return value 0 from its read() call, which will be delivered when the writer closes the file after it is finished writing. In this case the writer is the process that executes "ls -l". When the output of ls -l is written to the pipe, cat will read it and display it on the screen. The wait command's only purpose is to delay the shell's prompt until after cat exits.

By the way, if you reverse this procedure:

```
$ ls -l > PIPE &
$ ls -l PIPE
$ cat < PIPE; wait
```

and expect to see that the PIPE does not have 0 bytes when the second ls -l is executed, you will be disappointed. That data is not stored in the file system.

8.4.2 Programming With Named Pipes

We turn to the creation and use of named pipes at the programming level. A named pipe can be created either by using the mknod() system call, or the mkfifo() library function. In Linux, according to the mknod() (2) man page,

"*Under Linux, this call cannot be used to create directories. One should make directories with mkdir(2), and FIFOs with mkfifo(3).*"

Therefore, we will stick to using mkfifo() for creating FIFOs. The other advantage of mkfifo() over mknod() is that it is easier to use and does not require superuser privileges:

```
#include <sys/types.h>
#include <sys/stat.h>
int mkfifo(const char *pathname, mode_t mode);
```

The call mkfifo("MY_PIPE", 0666) creates a FIFO named MY_PIPE with permission 0666 & ~umask. The convention is to use UPPERCASE letters for the names of FIFOs. This way they are easily identified in directory listings.

It is useful to distinguish between *public* and *private* FIFOs. A *public FIFO* is one that is known to all clients. It is not that there is a specific function that makes a FIFO public; it is just that it is given a name that is easy to remember and that its location is advertised so that client programs know where to find it. Some authors call these *well-known FIFOs*, because they are analogous to *well-known ports* used for sockets, which are covered later. A *private FIFO*, in contrast, is given a name that is not known to anyone except the process that creates it and the processes to which it chooses to divulge it. In our first example, we will use only a single public FIFO. In the second example, the server will create a public FIFO and the clients will create private FIFOs that they will each use exclusively for communicating with the server.

8.4.2.1 Example

This is a simple example that demonstrates the basic principles. In it, the server creates a public FIFO and listens for incoming messages. When a message is received, it just prints it on the console. Client programs know the name of the FIFO because its pathname is hard-coded into a publicly available header file that they can include. In fact, for this example, the server and the clients share this common header file. Ideally the FIFO's name should be chosen so that no other processes in the system would ever choose the same file name, but for simplicity, we use a name that may not be unique. [3].

The server will execute a loop of the form

```
while ( 1 ) {
    memset(buffer, 0, PIPE_BUF);
    if ( ( nbytes = read( publicfifo, buffer, PIPE_BUF)) > 0 ) {
        buffer[nbytes] = '\0';
        printf("Message %d received by server: %s", ++count, buffer);
        fflush(stdout);
    }
    else
         break;
}
```

In each iteration, it begins by zeroing the buffer into which it will copy the FIFO's contents. It reads at most `PIPE_BUF` bytes at a time into the buffer. When `read()` returns, if `nbytes` is positive, it null-terminates the buffer and writes what it received onto its controlling terminal. Because the input data may not have a terminating newline, it forces the write by calling `fflush(stdout)`. If `nbytes` is negative, there was an error and the server quits. If `nbytes` is 0, it means that `read()` returned without any data, and so there is nothing for it to write. We could design the loop so that it does not exit in this case but just re-executes the `read()`, but there are reasons not to, as we now explain.

The server has to perform blocking reads (the `O_NONBLOCK` and `O_NDELAY` flags are clear), otherwise it would continually run in a loop, needlessly calling `read()` until a client actually wrote to the FIFO. This would be a waste of CPU cycles. By using a blocking read, it relinquishes the CPU so that it can be used for other purposes. The problem is that the `read()` call will return 0 when there are no processes writing to the FIFO, so if no clients attempt to write to the server, or if all clients that were writing close their ends of the FIFO and exit, the server would receive a 0 from the `read()`. If we designed the loop so that it was re-entered when `read()` returned 0, this would not be a problem. However, it is a cleaner design to let the server open the FIFO for writing, so that

[3]There are programs that can generate unique keys of an extremely large size that can be used in the name of the file. If all applications cooperate and use this method, then all pipe names would be unique.

there is always at least one process holding the FIFO open for writing, and so that the return value of `read()` will be either positive or negative, unless there is some unanticipated error condition.

Therefore, the server begins by creating the FIFO and opening it for both reading and writing, even though it will only read from it. Since the server never writes to this pipe, it does not matter whether or not writes are non-blocking, but POSIX does not specify how a system is supposed to handle opening a file in blocking mode for both reading and writing, so it is safer to open it with the `O_NONBLOCK` flag set, since POSIX does not specify how a system is supposed to handle opening a file in blocking mode for both reading and writing, we avoid possibly undefined behavior.

The server is run as a background process and is the process that must be started first, so that it can create the FIFO. If the server is not running and a client is started up, it will exit, because the FIFO does not exist.

The common header file is listed first, in Listing 8.9, followed by the server code.

Listing 8.9: fifo1.h

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <fcntl.h>
#include <string.h>
#include <limits.h>
#include <errno.h>
#include <sys/stat.h>

#define  PUBLIC          "/tmp/FIFODEMO1_PIPE"
```

Listing 8.10: rcvfifo1.c

```
#include <signal.h>
#include "fifo1.h"

int    dummyfifo;       /* file descriptor to write-end of PUBLIC */
int    publicfifo;      /* file descriptor to read-end of PUBLIC */

/** on_signal()
 *  This closes both ends of the FIFO and then removes it, after
 *  which it exits the program.
 */
void on_signal( int sig )
{
    close(publicfifo);
    close(dummyfifo);
    unlink(PUBLIC);
    exit(0);
}

int main( int argc, char *argv[])
{
    int              nbytes;          /* number of bytes read from popen() */
    int              count = 0;
    static char      buffer[PIPE_BUF];/* buffer to store output of command */
    struct sigaction handler;         /* sigaction for registering handlers*/

    /* Register the signal handler to handle a few signals */
    handler.sa_handler = on_signal;       /* handler function     */
    handler.sa_flags = SA_RESTART;
    if ( ((sigaction(SIGINT, &handler, NULL)) == -1 ) ||
         ((sigaction(SIGHUP, &handler, NULL)) == -1 ) ||
         ((sigaction(SIGQUIT, &handler, NULL)) == -1) ||
         ((sigaction(SIGTERM, &handler, NULL)) == -1)
       ) {
            perror("sigaction");
            exit(1);
    }

    /* Create public FIFO. If it exists already, the call will return -1 and
       set errno to EEXIST. This is not an error in our case. It just means
       we can reuse an existing FIFO that we created but never removed. All
       other errors cause the program to exit.
    */
    if ( mkfifo(PUBLIC, 0666) < 0 )
        if (errno != EEXIST ) {
            perror(PUBLIC);
            exit(1);
        }

    /*
      We open the FIFO for reading, with the O_NONBLOCK flag clear. The POSIX
      semantics state the the process will be blocked on the open() until some
      process (to be precise, some thread) opens it for writing. Therefore, the
      server will be stuck in this open() until a client starts up.
    */
    if ( (publicfifo = open(PUBLIC, O_RDONLY) ) == -1 )  {
        perror(PUBLIC);
        exit(1);
    }

    /*
      We now open the FIFO for writing, even though we have no intention of
      writing to the FIFO.  We will not reach the call to open()
      until a client runs, but once the client runs, the server opens the FIFO
      for writing. If we do not do this, when the client terminates and closes
      its write-end of the FIFO, the server's read loop would exit and the
      server would also exit.  This "dummy" open keeps the server alive.
    */
    if ( (dummyfifo = open(PUBLIC, O_WRONLY | O_NONBLOCK )) == -1  ) {
        perror(PUBLIC);
        exit(1);
    }

    /* Block waiting for a message from a client */
    while ( 1 ) {
        memset(buffer, 0, PIPE_BUF);
        if ( ( nbytes = read( publicfifo, buffer, PIPE_BUF)) > 0 ) {
            buffer[nbytes] = '\0';
            printf("Message %d received by server: %s", ++count, buffer);
            fflush(stdout);
        }
        else
             break;
    }
    return 0;
}
```

Comments.

- The server reads from the public FIFO and displays the message it receives on its standard output, even though it may be put in the background; it is not detached from the terminal. The best way to run it is to leave it in the foreground and open a few clients in other terminal windows.
- The server increments a counter and displays each received message with the value of the counter, so that you can see the order in which the messages were received. As noted above, it flushes standard output just in case there is no newline in the message.
- It does detect a few signals, so that any of them are delivered to it, it will close its ends of the FIFO, remove the file, and bail out.

The client opens the public FIFO for writing and then enters a loop where it repeatedly reads from standard input and writes into the write-end of the public FIFO. It uses the library function `memset()`, found in `<string.h>`, to zero the buffer where the user's text will be stored, and it declares the buffer to be `PIPE_BUF` chars, so that the write will be atomic. (If the locale uses two-byte chars, this will not work properly.) When it is finished, it closes its write-end.

Listing 8.11: sendfifo1.c

```
#include "fifo1.h"
#define   QUIT          "quit"

int main( int argc, char *argv[])
{
    int     nbytes;          /* num bytes read */
    int     publicfifo;      /* file descriptor to write-end of PUBLIC */
    char    text[PIPE_BUF];


    /* Open the public FIFO for writing */
    if ( (publicfifo = open(PUBLIC, O_WRONLY) ) == -1) {
        perror(PUBLIC);
        exit(1);
    }

    printf("Type 'quit' to quit.\n");

    /* Repeatedly prompt user for command, read it, and send to server */
    while (1) {
        memset(text, 0, PIPE_BUF);                  /* zero string */
        nbytes = read(fileno(stdin), text, PIPE_BUF);
        if ( !strncmp(QUIT, text, nbytes-1))        /* is it quit? */
            break;

        if ( ( write(publicfifo, text, nbytes)) < 0 ) {
            perror("Server is no longer running");
            break;
        }
    }
    /* User quit, so close write-end of public FIFO */
    close(publicfifo);
    return 0;
}
```

Comments.

- The client code allows the user to type `"quit"` to end the program.
- It is not very robust; it does not handle any terminal interrupts or signals and does no clean-up if it is killed by a signal. If the server stops running though, it will detect this and exit, closing its end of the FIFO.

8.4.3 An Iterative Server

In this example, we create a server that has two way communication with each client, processing incoming client requests one after the other. Such a server is called an *iterative server*. In order to achieve this, the server creates a public FIFO that it uses for reading incoming messages from clients wishing to use its services. Each incoming message is a structure with a member that contains the name of the private FIFO that the client creates when it starts up, and which should be used by the server for sending a reply. The message structure also contains another field that the client can use to supply data for the server.

When the server receives a message, it looks at the FIFO name in it and tries to open it for writing. If successful, the server will use this FIFO for sending data to the client. After the client sends its message to the server, it opens its private FIFO for reading. It will block until the server opens the write end of this FIFO. When the server opens the write end, the client will read from it until it receives a return value of 0, indicating that the server has finished writing and closed its end of the pipe. Figure 8.6 depicts the relationship between the clients and the server with respect to the shared pipes.

In this particular example, the server provides lowercase-to-uppercase translation for clients. The clients send it a piece of text and the server sends back another piece of text identical to the first except that every lowercase letter has been converted to uppercase. The server will be named `upcased1` (for *uppercase daemon*), and the client, `upcaseclient1`.

The message structure used by the server and client, as well as all necessary include files and common definitions, is contained in the header file `upcase1.h`, displayed in the following listing.

Listing 8.12: upcase1.h

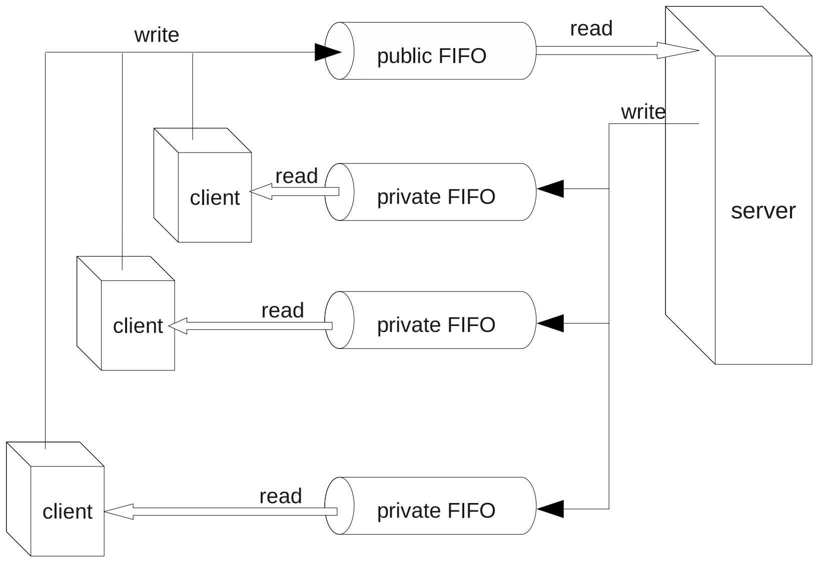

Figure 8.6: The FIFOs used in the iterative server.

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <fcntl.h>
#include <string.h>
#include <limits.h>
#include <signal.h>
#include <errno.h>
#include <sys/wait.h>
#include <sys/stat.h>
#include <ctype.h>

#define PUBLIC          "/tmp/UPCASE1_PIPE"
#define HALFPIPE_BUF    (PIPE_BUF/2)

typedef struct _message {
    char    fifo_name[HALFPIPE_BUF];   /* private FIFO pathname */
    char    text[HALFPIPE_BUF];        /* message text          */
} message;
```

Because the message must be no larger than `PIPE_BUF` bytes, and because it should be general enough to allow FIFO pathnames of a large size, the structure is split equally between the length of the FIFO name and the length of the text to be sent to the server. Thus, `HALFPIPE_BUF` is defined as one half of `PIPE_BUF` and used as the maximum number of bytes in the string to be translated.

We begin with the client code this time. The basic steps that the client takes are as follows.

1. It makes sure that neither standard input nor output is redirected.
2. It registers its signal handlers.
3. It creates its private FIFO in `/tmp`.

4. It tries to open the public FIFO for writing in non-blocking mode.
5. It enters a loop in which it repeatedly
   (a) reads a line from standard input, and
   (b) repeatedly
      i. gets the next `HALFPIPE_BUF-1` sized chunk in the input text,
      ii. sends a message to the server through the public FIFO,
      iii. opens its private FIFO for reading,
      iv. reads the server's reply from the private FIFO,
      v. copies the server's reply to its standard output, and
      vi. closes the read-end of its private FIFO.
6. It closes the write-end of the public FIFO and removes its private FIFO.

The client listing follows.

Listing 8.13: upcaseclient1.c

```
#include "upcase1.h"  /* All required header files are included in  */
                      /* this shared header file. */

#define  PROMPT   "string: "
#define  UPCASE   "UPCASE: "
#define  QUIT     "quit"

const char   startup_msg[] =
"upcased1 does not seem to be running. "
"Please start the service.\n";

volatile sig_atomic_t  sig_received = 0;
struct message   msg;

/*****************************************************************************/
/*                              Signal Handlers                              */
/*****************************************************************************/

void on_sigpipe( int signo )
{
    fprintf(stderr, "upcased is not reading the pipe.\n");
    unlink(msg.fifo_name);
    exit(1);
}

void on_signal( int sig )
{
    sig_received = 1;
}
/*****************************************************************************/
/*                               Main Program                                */
/*****************************************************************************/
```

```
int main( int argc, char *argv[])
{
    int             strLength;      /* number of bytes in text to convert    */
    int             nChunk;         /* index of text chunk to send to server  */
    int             bytesRead;      /* bytes received in read from server     */
    int             privatefifo;    /* file descriptor to read-end of PRIVATE */
    int             publicfifo;     /* file descriptor to write-end of PUBLIC */
    static char     buffer[PIPE_BUF];
    static char     textbuf[BUFSIZ];

    struct sigaction handler;

    /* Only run if we are using the terminal. */
    if ( !isatty(fileno(stdin)) || !isatty(fileno(stdout)) )
        exit(1);

    /* Register the on_signal handler to handle all keyboard signals */
    handler.sa_handler = on_signal;         /* handler function    */
    if ( ((sigaction(SIGINT, &handler, NULL)) == -1 ) ||
         ((sigaction(SIGHUP, &handler, NULL)) == -1 ) ||
         ((sigaction(SIGQUIT, &handler, NULL)) == -1) ||
         ((sigaction(SIGTERM, &handler, NULL)) == -1)
       ) {
            perror("sigaction");
            exit(1);
    }

    handler.sa_handler = on_sigpipe;
    if ( sigaction(SIGPIPE, &handler, NULL) == -1 ) {
            perror("sigaction");
            exit(1);
    }

    /* Create hopefully unique name for private FIFO using process-id */
    sprintf(msg.fifo_name, "/tmp/fifo%d",getpid());

    /* Create the private FIFO  */
    if ( mkfifo(msg.fifo_name, 0666) < 0 ) {
        perror(msg.fifo_name);
        exit(1);
    }

    /* Open the public FIFO for writing */
    if ( (publicfifo = open(PUBLIC, O_WRONLY | O_NONBLOCK) ) == -1) {
        if ( ENXIO == errno )
            fprintf(stderr,"%s", startup_msg);
        else
            perror(PUBLIC);
        exit(1);
    }
    printf("Type 'quit' to quit.\n");

    /* Repeatedly prompt user for input, read it, and send to server */
    while (1) {
        /* Check if SIGINT received first, and if so, close write-end */
        /* of public fifo, remove private fifo and then quit */
        if ( sig_received ) {
            close(publicfifo);
            unlink(msg.fifo_name);
            exit(1);
        }

        /* Display a prompt on the terminal and read the input text */
        write( fileno(stdout), PROMPT, sizeof(PROMPT));
        memset(msg.text, 0x0, HALFPIPE_BUF);          /* zero string */
        fgets(textbuf, BUFSIZ, stdin);
        strLength = strlen(textbuf);
        if ( !strncmp(QUIT, textbuf, strLength-1))    /* is it quit? */
            break;

        /* Display label for returned upper case text */
        write(fileno(stdout), UPCASE, sizeof(UPCASE));

        for ( nChunk = 0; nChunk < strLength; nChunk += HALFPIPE_BUF-1 ) {
            memset(msg.text, 0x0, HALFPIPE_BUF);
            strncpy(msg.text, textbuf+nChunk, HALFPIPE_BUF-1);
            msg.text[HALFPIPE_BUF-1] = '\0';
            write(publicfifo, (char*) &msg, sizeof(msg));

            /* Open the private FIFO for reading to get output of command */
            /* from the server. */
            if ((privatefifo = open(msg.fifo_name, O_RDONLY) ) == -1) {
                perror(msg.fifo_name);
                exit(1);
            }

            /* Read maximum number of bytes possible atomically */
            /* and copy them to standard output. */
            while ((bytesRead= read(privatefifo, buffer, PIPE_BUF)) > 0) {
                write(fileno(stdout), buffer, bytesRead);
            }
            close(privatefifo);      /* close the read-end of private FIFO */
        }
    }
    /* User quit, so close write-end of public FIFO and delete private FIFO */
    close(publicfifo);
    unlink(msg.fifo_name);
    return 0;
}
```

Comments.

- The program registers `on_signal()` to handle all signals that could kill it and that can be generated by a user. If any of these signals is sent to the process, the handler simply sets an atomic flag. In its main loop, it checks whether the flag is set, and if it is, it closes the write-end of the public FIFO and removes its private FIFO. The server will get a `SIGPIPE` signal the next time it tries to write to this FIFO, which it will handle.

- The program will get a `SIGPIPE` signal if it tries to write to the public FIFO but it is not open for reading. This can only happen if the server is not running. The `SIGPIPE` handler, `on_sigpipe()`, displays a message on standard error and terminates the program.
- The reason that the client opens the public FIFO with `O_NONBLOCK` set is that, in this case, if the server is not reading the FIFO, the client, instead of blocking, will return with a `ENXIO` error, so that it can gracefully exit.
- Inside the client's main loop, it displays a prompt and uses `fgets()` to read a line from the terminal.
- This client has been designed to handle the highly improbable case that the user enters a string that is larger than the allowed number of bytes in an atomic write to a pipe[4]. It does this by breaking the string into "chunks" that are small enough to send atomically. It send each chunk in sequence. It has to open and close the private FIFO before and after each chunk is sent because the server is designed primarily for handling the most likely case in which the string is small enough to fit into a single chunk. (The server only opens the client's private FIFO after receiving a message from the client with the name of the FIFO; if the client tries to open the FIFO for reading before sending any chunks, it will block on the `open()` call. To prevent this, the `open()` would have to be non-blocking, which would complicate its read loop. It is not worth the complication to save the run-time cost in this unusual case.)

Now we turn to the server, which is simpler than the client in this example. The steps that the server takes can be summarized as follows.

1. It registers its signal handlers.
2. It creates the public FIFO. If it finds it already exists, it displays a message and exits.
3. It opens the public FIFO for both reading and writing, even though it will only read from it.
4. It enters its main-loop, where it repeatedly
   (a) does a blocking read on the public FIFO,
   (b) on receiving a message from the `read()`, tries to open the private FIFO of the client that sent it the message. (It tries 5 times, sleeping a bit between each try, in case the client was delayed in opening it for writing. After 5 attempts it gives up on this client.)
   (c) converts the message to uppercase,
   (d) writes it to the private FIFO of the client, and
   (e) closes the write-end of the private FIFO.

It will loop forever because it will never receive an end-of-file on the pipe, since it is keeping the write-end open itself. It is terminated by sending it a signal. The code follows.

[4]Since `BUFSIZ`, the maximum size string allowed in the Standard I/O Library, may be larger than `PIPE_BUF`, it is possible to read a string much larger than can be sent in the pipe atomically.

Listing 8.14: upcased1.c

```
#include "upcase1.h"

#define  WARNING  "\nNOTE: SERVER ** NEVER ** accessed private FIFO\n"
#define  MAXTRIES 5

int          dummyfifo;    /* file descriptor to write-end of PUBLIC */
int          privatefifo;  /* file descriptor to write-end of PRIVATE */
int          publicfifo;   /* file descriptor to read-end of PUBLIC */

void on_sigpipe( int signo )
{
    fprintf(stderr, "Client is not reading the pipe.\n");
}

void on_signal( int sig )
{
    close(publicfifo);
    close(dummyfifo);
    if ( privatefifo != -1 )
        close(privatefifo);
    unlink(PUBLIC);
    exit(0);
}

/****************************************************************************/
/*                              Main Program                                */
/****************************************************************************/

int main( int argc, char *argv[])
{
    int               tries;        /* num tries to open private FIFO */
    int               nbytes;       /* number of bytes read from popen()  */
    int               i;
    int               done;         /* flag to stop loop */
    struct message    msg;          /* stores private fifo name and command */
    struct sigaction handler;       /* sigaction for registering handlers */


    /* Register the signal handler  */
    handler.sa_handler = on_signal;
    handler.sa_flags = SA_RESTART;
    if ( ((sigaction(SIGINT, &handler, NULL)) == -1 ) ||
         ((sigaction(SIGHUP, &handler, NULL)) == -1 ) ||
         ((sigaction(SIGQUIT, &handler, NULL)) == -1) ||
         ((sigaction(SIGTERM, &handler, NULL)) == -1)
       ) {
            perror("sigaction");
            exit(1);
    }

    handler.sa_handler = on_sigpipe;
    if ( sigaction(SIGPIPE, &handler, NULL) == -1 ) {
            perror("sigaction");
            exit(1);
    }

    /* Create public FIFO */
    if ( mkfifo(PUBLIC, 0666) < 0 ) {
        if (errno != EEXIST )
            perror(PUBLIC);
        else
            fprintf(stderr, "%s already exists. Delete it and restart.\n",
                    PUBLIC);
        exit(1);
    }


    /* Open public FIFO for reading and writing so that it does not get an
       EOF on the read-end while waiting for a client to send data.
       To prevent it from hanging on the open, the write-end is opened in
       non-blocking mode. It never writes to it.
    */
    if ( (publicfifo = open(PUBLIC, O_RDONLY) ) == -1 ||
         ( dummyfifo = open(PUBLIC, O_WRONLY | O_NDELAY )) == -1   ) {
        perror(PUBLIC);
        exit(1);
    }

    /* Block waiting for a msg struct from a client */
    while ( read( publicfifo, (char*) &msg, sizeof(msg)) > 0 ) {
        /* A msg arrived, so start trying to open write end of private FIFO */
        tries = done = 0;
        privatefifo = -1;
        do {
            if ( (privatefifo = open(msg.fifo_name,
                                 O_WRONLY | O_NDELAY)) == -1 )
                sleep(1);   /* sleep if failed to open */
            else {
                /* Convert the text to uppercase  */
                nbytes = strlen(msg.text);
                for ( i = 0; i < nbytes; i++ )
                    if ( islower(msg.text[i]))
                        msg.text[i] = toupper(msg.text[i]);

                /* Send converted text to client  */
                if ( -1 == write(privatefifo, msg.text, nbytes) ) {
                    if ( errno == EPIPE )
                        done = 1;
                }
                close(privatefifo);  /* close write-end of private FIFO */
                done = 1;            /* terminate loop */
            }
        } while (++tries < MAXTRIES && !done);

        if ( !done)
            /* Failed to open client private FIFO for writing */
            write(fileno(stderr), WARNING, sizeof(WARNING));
    }
    return 0;
}
```

Comments.

This server handles all user-initiated terminating signals by closing any descriptors that it has open and removing the public FIFO and exiting. It sets `privatefifo` to -1 at the start of each loop, and if it opens the private FIFO successfully, `privatefifo` is no longer -1. This way, in the signal handler, it can determine whether it had a private FIFO open for writing and needs to close it.

If it gets a `SIGPIPE` because a client closed its read end of its private FIFO immediately after sending a message but before the server wrote back the converted string, it handles `SIGPIPE` by continuing to listen for new messages and giving up on the write to that pipe.

8.4.4 Concurrent Servers

The preceding server was an iterative server; it handled each client request one after the other. If some client requests could be very time-consuming, then the server would be busy servicing one client to the exclusion of all others, and the others would experience delays. This can be avoided by allowing the server to handle multiple clients simultaneously. A server that can process requests from more than one client simultaneously is called a *concurrent server*.

The easiest way to create a concurrent server is to fork a child process for each client[5]. The server's role then amounts to little more than "listening" to the public pipe for incoming requests, forking a child process to handle a new request, and waiting for its children to finish. The waiting must be accomplished through a `SIGCHLD` handler, because, unlike a shell-style application, this process has to return immediately to the task of reading the public pipe. The basic outline of the server program's main process is therefore roughly:

1. It registers its signal handlers.
2. It creates the public FIFO. If it finds it already exists, it displays a message and exits.
3. It opens the public FIFO for both reading and writing, even though it will only read from it.
4. It enters its main-loop, where it repeatedly
   (a) does a blocking `read()` on the public FIFO,
   (b) on receiving a message from the `read()`, forks a child process to handle the client request.

Aside from spawning child processes, there are a few major differences between the way this server works and the way the sequential server worked:

Each client will have two private FIFOs: one into which it writes raw text to be translated, and a second from which it reads text that the server translated and sent back to it. The names of these two FIFOs must be sent to the server's public FIFO when a client wishes its services. Therefore,

[5]When we cover threads, you will see that threads are another means of accomplishing this.

the message structure is different in this program than it was in the iterative server. We will call this message a *connection message*, because its only purpose is to establish the means by which the client and the server can communicate privately. A connection message contains only the information needed to establish this two-way private communication between the server and the client:

```
typedef struct _message {
    char raw_text_fifo [HALFPIPE_BUF];
    char converted_text_fifo[HALFPIPE_BUF];
} message;
```

Each child process forked by the server begins by opening the read-end of the client's "raw_text" FIFO, and then it repeatedly reads from the this raw_text FIFO, translates the text into uppercase, opens the write-end of the client's converted_text FIFO, writes the converted text into it, and closes the write-end of the converted_text FIFO, until it received an end-of-file from the client.

8.4.4.1 The Concurrent Server Client

The client is also structurally different from the previous client. The major steps that it takes are as follows.

1. It registers its signal handlers.
2. It creates two private FIFOs in the `/tmp` directory with unique names.
3. It opens the server's public FIFO for writing.
4. It sends the initial message structure containing the names of its two FIFOs to the server to establish the two-way communication.
5. It attempts to open its raw_text FIFO in non-blocking, write-only mode. If it fails, it delays a second and retries. It retries a few times and then gives up and exits. If it fails it means that the server is probably terminated.
6. Until it receives an end-of-file on its standard input, it repeatedly
   (a) reads a line from standard input,
   (b) breaks the line into `PIPE_BUF`-sized chunks,
   (c) sends each chunk successively to the server through its raw_text FIFO,
   (d) opens the converted_text FIFO for reading,
   (e) reads the converted_text FIFO, and copies its contents to its standard output, and
   (f) closes the read-end of the converted_text FIFO
7. It closes all of its FIFOs and removes the files.

Figure 8.7 shows how the client processes and the server parent and child processes use the various FIFOs. Compare this to Figure 8.6.

The code for the client is displayed first, in the following Listing.

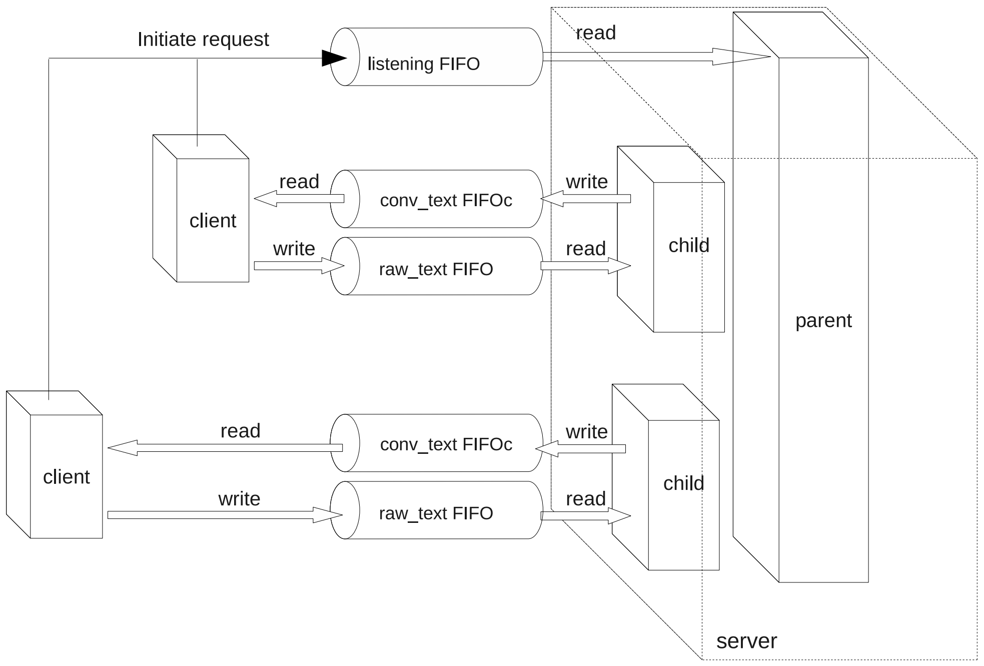

Figure 8.7: Concurrent server and client communication.

Listing 8.15: upcaseclient2.c

```
#include "upcase2.h"  /* All required header files are included in this */
                      /* shared header file.  */

#define  MAXTRIES 5

const char   startup_msg[] =
        "The upcased2 server does not seem to be running. "
        "Please start the service.\n";

const char   server_no_read_msg[] =
        "The server is not reading the pipe.\n";

int          convertedtext_fd; /* file descriptor for READ PRIVATE FIFO  */
int          dummyreadfifo;    /* to hold fifo open                      */
int          rawtext_fd;       /* file descriptor to WRITE PRIVATE FIFO  */
int          dummyrawfifo_fd;  /* to hold the raw text fifo open         */
int          publicfifo;       /* file descriptor to write-end of PUBLIC */
FILE*        input_srcp;       /* File pointer to input stream           */
message      msg;              /* 2-way communication structure          */

/*  Signal Handlers and Utilities  */

void on_sigpipe( int signo )
{
    fprintf(stderr, "upcased is not reading the pipe.\n");
    unlink(msg.raw_text_fifo);
    unlink(msg.converted_text_fifo);
    exit(1);
}

void on_signal( int sig )
{
    close(publicfifo);
    if ( convertedtext_fd != -1 )
        close(convertedtext_fd);
    if ( rawtext_fd != -1 )
        close(rawtext_fd);
    unlink(msg.converted_text_fifo);
    unlink(msg.raw_text_fifo);
    exit(0);
}

void clean_up()
{
    close(publicfifo);
    close(rawtext_fd);
    unlink(msg.converted_text_fifo);
    unlink(msg.raw_text_fifo);
}

/****************************************************************************/
/*                              Main Program                                */
/****************************************************************************/

int main( int argc, char *argv[])
{
    int              strLength;    /* number of bytes in text to convert   */
    int              nChunk;       /* index of text chunk to send to server*/
    int              bytesRead;    /* bytes received in read from server   */
    static char      buffer[PIPE_BUF];
    static char      textbuf[BUFSIZ];
    struct sigaction handler;
    int              tries;     /* for counting tries to open rawtext fifo */

    /* Check whether there is a command line argument, and if so, use it as
       the input source. */
    if ( argc > 1 ) {
        if ( NULL == (input_srcp  = fopen(argv[1], "r")) ) {
            perror(argv[1]);
            exit(1);
        }
    }
    else
        input_srcp = stdin;

    /* Initialize the file descriptors for error handling */
    publicfifo = -1;
    convertedtext_fd   = -1;
    rawtext_fd = -1;
    /* Register the on_signal handler to handle all signals */
    handler.sa_handler = on_signal;
    if ( ((sigaction(SIGINT, &handler, NULL)) == -1 ) ||
         ((sigaction(SIGHUP, &handler, NULL)) == -1 ) ||
         ((sigaction(SIGQUIT, &handler, NULL)) == -1) ||
         ((sigaction(SIGTERM, &handler, NULL)) == -1)
       ) {
        perror("sigaction");
        exit(1);
    }

    handler.sa_handler = on_sigpipe;
    if ( sigaction(SIGPIPE, &handler, NULL) == -1 ) {
        perror("sigaction");
        exit(1);
    }

    /* Create unique names for private FIFOs using process-id */
    sprintf(msg.converted_text_fifo, "/tmp/fifo_rd%d",getpid());
    sprintf(msg.raw_text_fifo, "/tmp/fifo_wr%d",getpid());

    /* Create the private FIFOs */
    if ( mkfifo(msg.converted_text_fifo, 0666) < 0 ) {
        perror(msg.converted_text_fifo);
        exit(1);
    }

    if ( mkfifo(msg.raw_text_fifo, 0666) < 0 ) {
        perror(msg.raw_text_fifo);
        exit(1);
    }

    /* Open the public FIFO for writing */
    if ( (publicfifo = open(PUBLIC, O_WRONLY | O_NDELAY) ) == -1) {
        if ( errno == ENXIO )
            fprintf(stderr,"%s", startup_msg);
        else
            perror(PUBLIC);
        exit(1);
    }

    /* Send a message to server with names of two FIFOs */
    write(publicfifo, (char*) &msg, sizeof(msg));

    /* Try to open the raw text FIFO for writing. After MAXTRIES
       attempts we give up. */
    tries = 0;
    while (((rawtext_fd = open(msg.raw_text_fifo,
             O_WRONLY | O_NDELAY )) == -1 ) && (tries < MAXTRIES ))   {
            sleep(1);
            tries++;
    }
    if ( tries == MAXTRIES ) {
        fprintf(stderr, "%s", server_no_read_msg);
        clean_up();
        exit(1);
    }

    /* Get one line of input at a time from the input source */
    while (1) {
        memset(textbuf, 0x0, BUFSIZ);
        if ( NULL == fgets(textbuf, BUFSIZ, input_srcp) )
            break;

        strLength = strlen(textbuf);

        /* Break input lines into chunks and send them one at a */
        /* time through the client's write FIFO */
        for ( nChunk = 0; nChunk < strLength; nChunk += PIPE_BUF-1 ) {
            memset(buffer, 0x0, PIPE_BUF);
            strncpy(buffer, textbuf+nChunk, PIPE_BUF-1);
            buffer[PIPE_BUF-1] = '\0';
            write(rawtext_fd, buffer, strlen(buffer));

            /* Open the private FIFO for reading to get output of command */
            /* from the server. */
            if ((convertedtext_fd = open(msg.converted_text_fifo, O_RDONLY) )
                  == -1) {
                perror(msg.converted_text_fifo);
                exit(1);
            }
            memset(buffer, 0x0, PIPE_BUF);
            while ((bytesRead= read(convertedtext_fd, buffer, PIPE_BUF)) > 0)
                write(fileno(stdout), buffer, bytesRead);

            close(convertedtext_fd);
            convertedtext_fd   = -1;
        }
    }
    /* User quit, so close write-end of public FIFO and delete private FIFO */
    close(publicfifo);
    close(rawtext_fd);
    unlink(msg.converted_text_fifo);
    unlink(msg.raw_text_fifo);
    return 0;
}
```

Comments.

- The order of events here is important, and in some cases critical. After the client creates its private FIFOs without error, it opens the write-end of the server's public FIFO. It then sends a message containing the names of its private FIFOs. After sending the names of the private FIFOs, it tries to open the write-end of its raw_text FIFO in non-blocking mode. This will fail if the server has not opened the read-end yet. Assuming that the server is running, the client will succeed in opening the raw_text FIFO. The server can open its read-end without the write-end being open, so this works well. If we were to reverse the order and open the raw_text FIFO before sending the server the message, we would need to open it in read-write mode since the server is blocked on its read of the public FIFO and the two processes would deadlock otherwise. But if we open the raw_text FIFO in read-write mode, then if the server terminates unexpectedly and never reads the raw_text FIFO again, the client will not get a `SIGPIPE` signal because the client itself has a read-end open, preventing the kernel from generating the signal. The client would never be notified that the server died.

- The client then keeps the write-end of its raw_text FIFO open for the duration of its main loop.
- Within the loop, the client first writes to its raw_text FIFO, and then opens its converted_text FIFO, after which, if all goes well, it reads and closes it again. Thus, it repeatedly opens and closes this FIFO within the loop. We could just let it stay open for the duration of the loop, but closing it and re-opening it we give ourselves the chance to detect in the `open()` call that the server closed its write end of the FIFO unexpectedly.
- The error handling in the client is similar to what it was in the iterative server's client. The code has redundant error checks such as guards to prevent closing a FIFO that is not open (setting the file descriptors to -1 unless they are in use), and closing descriptors before unlinking the files. On the other hand, it should really check the return values of the `close()` calls. A `clean_up()` function simplifies the error-handling, consolidating the cleaning up code.

8.4.4.2 The Concurrent Server

The server code is in the next listing.

Listing 8.16: upcased2.c

```
#include "upcase2.h"
#include "sys/wait.h"

#define  WARNING    "Server could not access client FIFO\n"
#define  MAXTRIES 5

int     dummyfifo;            /* file descriptor to write-end of PUBLIC */
int     client_convertedtext_fd; /* file descriptor to write-end of PRIVATE */
int     client_rawtext_fd;    /* file descriptor to write-end of PRIVATE */
int     publicfifo;           /* file descriptor to read-end of PUBLIC */
FILE*   upcaselog_fp;         /* points to log file for server */
pid_t   server_pid;           /* to store server's process id */

/*****************************************************************************/
/*                        Signal Handler Prototypes                          */
/*****************************************************************************/
/** on_sigpipe()
 *  This handles the SIGPIPE signals, just writes to standard error.
 */
void on_sigpipe( int signo );

/** on_signal()
 *  This handles the interrupt signals. It closes open FIFOs and files,
 *  removes the public FIFO and exits.
 */
void on_signal( int sig );

/** on_sigchild()
 *  Because this is a concurrent server, the parent process has to collect the
 *  exit status of each child. The SIGCHLD handler issues waits and writes to
 *  the log file.
 */
void on_sigchld( int signo );

/****************************************************************************/
/*                              Main Program                                */
/****************************************************************************/

int main( int argc, char *argv[])
{
    int               tries;      /* num tries to open private FIFO  */
    int               nbytes;     /* number of bytes read from private FIFO */
    int               i;
    struct message    msg;        /* message structure with FIFO names */
    struct sigaction  handler;    /* sigaction for registering handlers */
    char              buffer[PIPE_BUF];
    char              logfilepath[PATH_MAX];
    char              *homepath;      /* path to home directory */
    pid_t             child_pid;      /* pid of each spawned child */
    /* Open the log file in the user's home directory for appending. */
    homepath  = getenv("HOME");
    sprintf(logfilepath, "%s/.upcase_log", homepath );

    if ( NULL == (upcaselog_fp  = fopen(logfilepath, "a")) ) {
        perror(logfilepath);
        exit(1);
    }

    /* Register the interrupt signal handler  */
    handler.sa_handler = on_signal;
    handler.sa_flags = SA_RESTART;
    if ( ((sigaction(SIGINT, &handler, NULL)) == -1 ) ||
         ((sigaction(SIGHUP, &handler, NULL)) == -1 ) ||
         ((sigaction(SIGQUIT, &handler, NULL)) == -1) ||
         ((sigaction(SIGTERM, &handler, NULL)) == -1)
       ) {
        perror("sigaction");
        exit(1);
    }

    handler.sa_handler = on_sigpipe;
    if ( sigaction(SIGPIPE, &handler, NULL) == -1 ) {
        perror("sigaction");
        exit(1);
    }

    handler.sa_handler = on_sigchld;
    if ( sigaction(SIGCHLD, &handler, NULL) == -1 ) {
        perror("sigaction");
        exit(1);
    }

    /* Create public FIFO */
    if ( mkfifo(PUBLIC, 0666) < 0 ) {
        if (errno != EEXIST )
            perror(PUBLIC);
        else  {
            fprintf(stderr, "%s already exists. Delete it and restart.\n",
                    PUBLIC);
        }
        exit(1);
    }

    /* Open public FIFO for reading and writing so that it does not get */
    /* EOF on the read-end while waiting for a client to send data.     */
    /* To prevent it from hanging on the open, the write-end is opened  */
    /* in non-blocking mode. It never writes to it. */
    if ( (publicfifo = open(PUBLIC, O_RDONLY) ) == -1 ||
         ( dummyfifo = open(PUBLIC, O_WRONLY | O_NDELAY )) == -1  ) {
        perror(PUBLIC);
        exit(1);
    }

    server_pid = getpid();

    /* Block waiting for a msg structure from a client */
    while ( read( publicfifo, (char*) &msg, sizeof(msg)) > 0 ) {

        /* spawn child process to handle this client */
        if ( 0 == fork() ) {

            /* We get the pid for message identification. */
            child_pid = getpid();

            /* We use the value of client_rawtext_fd to detect errors */
            client_rawtext_fd = -1;

            /* Client should have opened rawtext_fd for writing before
               sending the message structure, so the following open should
               succeed immediately. If not it blocks until the client opens
               it.  */
            if ( (client_rawtext_fd = open(msg.raw_text_fifo, O_RDONLY))
                  == -1 ) {
                 fprintf(upcaselog_fp,
                         "Client did not have pipe open for writing\n");
                 exit(1);
            }
            /* Clear the buffer used for reading the client's text */
            memset(buffer, 0x0, PIPE_BUF);

            /*
              Attempt to read from client's raw_text_fifo. This read will
```

```
               block until either input is available or it receives an EOF.
               An EOF is delivered only when the client closes the write-end
               of its raw_text_fifo.
            */
            while ( (nbytes = read(client_rawtext_fd, buffer,
                    PIPE_BUF)) > 0 ) {
                /* Convert the text to uppercase */
                for ( i = 0; i < nbytes; i++ )
                    if ( islower(buffer[i]))
                        buffer[i] = toupper(buffer[i]);

                /* Open client's convertedtext FIFO for writing. To allow for
                   delays, we try 5 times. Here it is critical that the
                   O_NONBLOCK flag is set, otherwise it will hang in the loop
                   and we will not be able to abandon the attempt if the client
                   has died.  */
                tries = 0;
                while (((client_convertedtext_fd = open(msg.converted_text_fifo,
                        O_WRONLY | O_NDELAY)) == -1 ) && (tries < MAXTRIES ))
                {
                     sleep(2);
                     tries++;
                }
                if ( tries == MAXTRIES ) {
                    /* Failed to open client convertedtext FIFO for writing */
                    fprintf(upcaselog_fp, "%d: " WARNING, child_pid);
                    exit(1);
                }

                /* Send converted text to client in its readfifo */
                if ( -1 == write(client_convertedtext_fd, buffer,
                     nbytes) ) {
                    if ( errno == EPIPE )
                        exit(1);
                }
                /* See the notes below. */
                close(client_convertedtext_fd);
                client_convertedtext_fd = -1;

                /* Clear the buffer used for reading the client's text */
                memset(buffer, 0x0, PIPE_BUF);
            }
            exit(0);
        }
    }
    return 0;
}
```

The signal handlers for the server are below. The `SIGCHLD` handler uses `waitpid()` to wait for all children, and it remains in its loop as long as there is a zombie to be collected. The `WNOHANG` flag is used to prevent it from blocking in the `waitpid()` code. This way, if multiple `SIGCHLD` signals arrive while it is in the handler, the children whose deaths caused them will be collected. (Remember that signals may not be reliably handled on all systems, and even though in a POSIX compliant system, each SIGCHLD will be delivered if we set SA_NODEFER, it is safer to collect them in this loop.)

```
void on_sigchld( int signo )
{
    pid_t pid;
    int   status;

    while ( (pid = waitpid(-1, &status, WNOHANG) ) > 0 )
        fprintf(upcaselog_fp, "Child process %d terminated.\n", pid);
    fflush(upcaselog_fp);
    return;
}

void on_sigpipe( int signo )
{
    fprintf(stderr, "Client is not reading the pipe.\n");
}

void on_signal( int sig )
{
    close(publicfifo);
    close(dummyfifo);
    if ( clientreadfifo != -1 )
        close(clientreadfifo);
    if ( clientwritefifo != -1)
        close(clientwritefifo);
    if ( getpid() == server_pid )
        unlink(PUBLIC);
    fclose(upcaselog);
    exit(0);
}
```

Comments.

- All of the work is performed by the child processes. Each child begins by trying to open the client's raw_text FIFO for reading. If successful, it enters a loop in which it repeatedly reads, converts the text to uppercase, opens the client's converted_text FIFO, writes the converted text to it, and closes it.
- Since the client may not have the converted_text open for reading for any number of reasons – it might have been terminated – the child process tries the `open()` a fixed number of times before it gives up. It uses the same technique as the iterative server did, using a non-blocking `open()`.
- When the child process does successfully open the FIFO, it still checks whether the `write()` failed, since anything can happen in between, and if so, the child exits. Otherwise, it writes the data, closes its end of the FIFO and waits to read more text from the client. When it receives the end-of-file, it exits.
- You may wonder why the server repeatedly opens and closes the write end of the client's converted_text FIFO. This is the only way that the client will receive an EOF in its `read()`.

If the client does not get the EOF, then it will remain blocked in its read of the converted_text FIFO, and will not be able to send any more data to the server. This would put the client and this server subprocess into deadlock, because this process would go back to the `read()` of the client's raw_text FIFO and block waiting for data from the client, which would never arrive. Therefore, although it seems inefficient to open and close this FIFO each time, it is the simplest means of preventing deadlock.

- The signal handler checks whether the parent process is executing it. If the parent has been signaled, then it should remove the public FIFO, otherwise not. We do not want child processes to remove this FIFO!
- If you are at all familiar with sockets, you might have noticed that the design of this server is easily converted to one that uses sockets. We will refer back to this example when we take up sockets.

8.5 Daemon Processes

As was mentioned earlier, a daemon is a process that runs in the background, has no controlling terminal. In addition, daemons set their working directory to "/". Usually daemons are started by system initialization scripts at boot-time. If you have written a server and want to turn it into a full-fledged daemon, it is not enough to put it into the background. This will only tell the shell not to wait for it; it will still have a control terminal and will still be killed by any signals from that terminal.

Some daemons are started by other programs. For example, some network daemons are started by the `inetd` or `xinetd` superserver. Some are started by programs such as the `crond` daemon, which runs scheduled jobs. Some are invoked at the user terminal. For example, sometimes the printer daemon is stopped and restarted at the terminal by the superuser.

Because daemons do not have a controlling terminal, they cannot write messages to standard output or to standard error. Instead they can use a system logging function named `syslog()`, which is a client that talks to the `syslogd` daemon, which write messages to specific log files. The `glibc` version of this function is `klogctl()`. Later we will look at an example of how it can be used. A server should be designed to turn itself into a daemon. In other words, when the server is run, it should take all of the steps necessary to become a daemon, which include:

1. Putting itself in the background. It does this by forking a new process and executing its code as the child and having the parent execute `exit()`. When the parent exits, the shell that started it collects its exit status and thinks the invoked program has terminated (which it has.) The child, which is now the server, is no longer in the foreground, but it is still controlled by the terminal.
2. Making itself a session leader. Recall that a process can detach itself from a terminal by becoming a session leader, but only processes that are neither session leaders nor process group leaders can do this. Since the server is now a child of the original process, it is neither, so it can call `setsid()`, which makes it a session leader of a new session and a group leader of a new process group.
3. Registering its intent to ignore `SIGHUP`.

4. Forking another child process, terminating in the parent again, and letting the new child, which is the grandchild of the original process, execute the server code. In some versions of UNIX, when a session leader opens a terminal device (which it may want to do sometimes), that terminal is automatically made the control terminal for the process. By running as the child of a session leader, the server is now immune from this eventuality. In Linux, a process can set the `O_NOCTTY` flag on `open()` to prevent this. The reason for ignoring `SIGHUP` is that when a session leader terminates, all of its children are sent a `SIGHUP`, which would otherwise kill them. Since the parent is a session leader, the child must ignore `SIGHUP`.
5. Changing the working directory to "/".
6. Clearing the umask.
7. Closing any open file descriptors.

A procedure for doing all of these steps, based on one from [Stevens], is below.

Listing 8.17: daemon_init.c

```
void daemon_init(const char *pname, int facility)
{
    int         i;
    pid_t     pid;

    if ( (pid = fork()) == -1) {
        perror("fork");
        exit(1);
    }
    else if (pid != 0)
        exit(0);        /* parent terminates  */

    /* Child continues from here */
    /* Detach itself and make itself a sesssion leader */
    setsid();

    /* Ignore SIGHUP */
    signal(SIGHUP, SIG_IGN);

    if ( (pid = fork()) == -1) {
        perror("fork");
        exit(1);
    }
    else if ( pid != 0 )
        exit(0);    /* First child terminates */

    /* Grandchild continues from here */
    chdir("/");          /* change working directory  */

    umask(0); /* clear our file mode creation mask  */

    /* Close all open file descriptors */
    for (i = 0; i < MAXFD; i++)
        close(i);
    /* Start logging with syslog() */
    openlog(pname, LOG_PID, facility);

}
```

The final version of the upcase server incorporates this function and turns itself into a daemon. The only changes required are to include this function into the code and insert the line

```
daemon_init(argv[0], 0);
```

before the first executable statement.

8.6 Multiplexed I/O With select

Imagine the situation in which a process has multiple sources of input open for reading, such as a set of pipes as well as the terminal. Suppose the process has to respond to commands typed at the terminal as well as display messages that are available in the pipes. This is what is meant by *multiplexed input*: when a process has to obtain input available from multiple sources simultaneously. One solution would be to make all of the reads non-blocking and to continually poll each descriptor to see if there is data ready for reading on it. Polling, though, has many drawbacks, as we have seen, the most important of which is that it is wasteful of the CPU resource.

Another alternative would be to use asynchronous reads on each descriptor. This is also possible, but quite messy to code, and has the drawback that it relies on signals which may not be handled properly or reliably.

It is for these reasons that the `select()` system call was developed[6]. Basically, the `select()` call allows a process to listen to multiple descriptors at once and to be notified when any of them have pending input or output. Roughly put, `select()` is given a set of masks of file descriptors, representing I/O devices or files in which the process is interested. When input or output is ready on any of them, the appropriate bits in these masks are set. The process can check the masks to see which I/O is ready and can then read or write the ready descriptors. The `select()` call works with any file descriptor, so that it can be used with files, pipes, FIFOs, devices, and sockets.

select() is fairly complex:

```
/* According to POSIX.1-2001 */
#include <sys/select.h>

/* According to earlier standards */
#include <sys/time.h>
#include <sys/types.h>
#include <unistd.h>

int select(int nfds, fd_set *readfds, fd_set *writefds,
           fd_set *exceptfds, struct timeval *timeout);
```

[6]There is a similar call named `poll()`.

The parameters have the following meanings:

`ndfs`	The number of file descriptors of potential interest.
`readfds`	The address of a file descriptor mask indicating which file descriptors the process is interested in *reading*.
`writefds`	The address of a file descriptor mask indicating which file descriptors the process is interested in *writing*.
`exceptfds`	The address of a file descriptor mask indicating which file descriptors the process is interested in checking for *out-of-band* data[7]. (Out-of-band messages or data should be thought of as exceptions or error conditions concerning any of the descriptors in the read or write descriptor masks.)
`timeout`	The address of a `timeval` structure containing the amount of time to wait before completing the `select()` call. If timeout is `NULL`, it means wait forever, i.e., block until at least one descriptor is ready. If it is zero, it means return immediately with the status of all descriptors in the above sets. If it is non-zero, it will either wait the specified amount of time or return before if one of the specified descriptors is ready.

The return value of the `select()` call is the number of descriptors that are ready, or -1 if there was an error.

The `fd_set` data type is not necessarily a scalar. It is usually an array of long integers. If you do a little digging you will discover a constant, `FD_SETSIZE`, that defines the maximum number of descriptors in a `fd_set`, which is usually on the order of 1024 or more. Fortunately, you do not need to know how it is defined to use it, since there are macros and/or functions in the library for manipulating `fd_set` objects:

This turns off the bit for descriptor `fd` in the mask pointed to by `fdset`:

```
void FD_CLR(int fd, fd_set *fdset);
```

This turns on the bit for descriptor `fd` in the mask pointed to by `fdset`:

```
void FD_SET(int fd, fd_set *fdset);
```

This sets all bits to zero in the mask pointed to by `fdset`:

```
void FD_ZERO(fd_set *fdset);
```

This checks whether the bit for descriptor `fd` is set in the mask pointed to by `fdset`:

```
int FD_ISSET(int fd, fd_set *set);
```

[7]Out-of-band refers to data that is transferred in a separate communication channel. Out-of-band implies that the data does not arrive in sequence with the rest of the data, but in a parallel channel. It is used for transmitting error or control messages.

The value of the first parameter, `ndfs`, must be set to the value of the largest file descriptor + 1, since the file descriptor array is 0-based. The reason that the first argument is the maximum number of descriptors of interest is for efficiency. By supplying this number to the kernel, it saves the kernel the work of having to copy parts of the descriptor mask that are not needed. To give you an idea of how this call is used in a simple case, if we wanted to read from two different open file descriptors, we would use something like

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

...

int     fd1, fd2, maxfd;
fd_set  readset, tempset;

fd1   = open("file1", O_RDONLY);  /* open file1  */
fd2   = open("file2", O_RDONLY);  /* open file2*/
maxfd = fd1 > fd2 ? fd1+1 : fd2+1;

FD_ZERO(&readset);         /* clear the bits in the mask  */
FD_SET(fd1, &readset);     /* set the bit for fd1 (file1) */
FD_SET(fd2, &readset);     /* set the bit for fd2 (file2) */
tempset = readset;         /* copy into tempset           */

while ( select(maxfd, &tempset, NULL, NULL, NULL) > 0) {
    if ( FD_ISSET(fd1, &tempset) ) {
         /* read from descriptor fd1 */
    }
    if ( FD_ISSET(fd2, &tempset) ) {
        /* read from descriptor fd2  */
    }
    tempset = readset;
}
```

Notes.

- Although we are interested only in file descriptors `fd1` and `fd2`, the proper way to use select is to specify the full range of descriptors from 0 to the maximum of `fd1` and `fd2`. Since it is a zero-based array, this value is `max(fd1, fd2) + 1`.
- Because the return value of `select()` is positive as long as there is data to be read on either of `fd1` or `fd2`, the loop will continue until we get end-of-file on both files.
- The way that `select()` works, it resets the file descriptor masks to reflect the status of the descriptors of interest. In other words, the masks change after each call to `select()`. Therefore, you need to keep a copy of the original mask, and before each call, reset the masks to their original states.
- The masks are not modified if the `select()` call returned with an error.

- Inside the loop, you use the `FD_ISSET()` function to test each descriptor in which you expressed interest.
- It is a very common mistake to forget to add 1 to the largest descriptor in the first argument. It is also a common mistake to forget to reset the mask between each successive call.

We will put these ideas to work in a slightly more interesting example, borrowed from *[Haviland et al]*.

Listing 8.18: selectdemo.c

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <limits.h>
#include <errno.h>
#include <sys/time.h>
#include <sys/wait.h>

#define  MSGSIZE   6

char  msg1[] = "Hello";
char  msg2[] = "Bye!!";

void parent( int pipeset[3][2]);
int child( int fd[2] );

/*****************************************************************************/
/*                              Main Program                                 */
/*****************************************************************************/

int main( int argc, char* argv[])
{
    int fd[3][2];  /* array of three pipes */
    int i;

    for ( i = 0; i < 3; i++ ) {
        /* create three pipes */
        if ( pipe(fd[i]) == -1 ) {
            perror("pipe");
            exit(1);
        }

        /* fork children */
        switch( fork() ) {
        case -1 :
            fprintf(stderr, "fork failed.\n");
            exit(1);
        case 0:
            child(fd[i]);
        }
    }
    parent(fd);
    return 0;
}

/****************************************************************************/
/*                              Parent and Child                            */
/****************************************************************************/

void parent( int pipeset[3][2])
{
    char     buf[MSGSIZE];
    char     line[80];
    fd_set   initial, copy;
    int      i, nbytes;

    for ( i = 0; i < 3; i++)
         close(pipeset[i][1]);

    /* create descriptor mask */
    FD_ZERO(&initial);
    FD_SET(0, &initial);                         /* add standard input */

    for ( i = 0; i < 3; i++)
        FD_SET(pipeset[i][0], &initial);         /* add read end of each pipe */

    copy = initial;                              /* make a copy */
    while (select( pipeset[2][0]+1, &copy, NULL, NULL, NULL ) > 0 ) {

        /* check standard input first */
        if ( FD_ISSET(0, &copy) ) {
            printf("From standard input: ");
            nbytes = read(0, line, 81);
            line[nbytes] = '\0';
            printf("%s", line);
        }

        /* check the pipe from each child */
        for ( i = 0; i < 3; i++ ) {
            if ( FD_ISSET(pipeset[i][0], &copy)) {
                /* it is ready to read */
                if ( read( pipeset[i][0], buf, MSGSIZE) > 0 ) {
                    printf("Message from child %d:%s\n", i, buf );
                }
            }
        }
        if (waitpid(-1, NULL, WNOHANG) == -1 )
            return;
        copy = initial;
    }
}

int child( int fd[2] )
{
    int count;
    close(fd[0]);

    for ( count = 0; count < 10; count ++) {
        write( fd[1], msg1, MSGSIZE);
        sleep(1 + getpid() % 6 );
    }

    write( fd[1], msg2, MSGSIZE);
    exit(0);
}
```

Comments.

- Each child writes a small string to the write-end of its pipe and then sleeps a bit so that the output does not flood the screen too quickly.
- The parent uses the `select()` call to query standard input and the read-ends of each child's pipe. The user can type a string on the keyboard and the parent will detect that standard input is ready. Within the while-loop each descriptor is tested, and if it is set, the **`read()`** can be done because input is waiting. This way the parent never holds up any child that is waiting for its message to be read.

There will be another, more interesting use of `select()` after the introduction to sockets.

8.7 Summary

Related processes can use unnamed pipes to exchange data. Unrelated processes running on the same host can use named pipes to exchange data. Unlike unnamed pipes, named pipes are entities in the file system. Both named and unnamed pipes are guaranteed by the kernel to be read and written atomically provide that the amount of data written is at most `PIPE_BUF` bytes.

Servers can be iterative or concurrent. A concurrent server creates a child process to handle every distinct client. An iterative server handles each client within a single process, sharing its time among them. Concurrent servers provide more reliable response time to the clients.

When a process has to handle I/O from multiple file descriptors, it can multiplex the I/O by means of the select() system call. This is one alternative of many, but it provides a relatively simple solution. Other alternatives include asynchronous I/O and the `poll()` call.