9 <u>Interprocess Communication (IPC)</u>

Concepts Covered

Pipes
I/O Redirection
FIFOs
Concurrent Servers
Daemons
Multiplexed I/O with select()
Sockets

mknod, tee,
API: *accept, bind, connect, dup, dup2, fpathconf, gethostbyname, listen, mkfifo, pipe, pclose, popen, recv, select, send, setsid, shutdown, socket, syslog,*

9.1 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.

We will begin with ***unnamed pipes***. After that we will look at ***named pipes***, also known as ***FIFO***'s, and then look at record-locking and file-locking schemes.

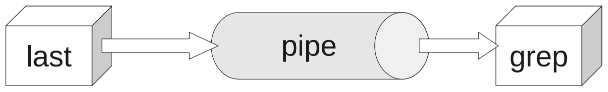
9.2 Unnamed Pipes

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`. Pipes were invented by Douglas Mcilroy. and were incorporated into UNIX in 1973.

We know that something called a *pipe* is involved in this mechanism; i.e., that there is some special file or buffer that we can visualize quite literally like a physical pipe, connecting the output of `last` to the input of `grep`:



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` 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 `demos/chapter09`) demonstrates this simple case.

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

#define  READ_END  0
```

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

```
#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];

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

    for ( i = 1; i <= NUM; i++ ) {
        sprintf(message, "hello #%2d", i);
        write(fd[WRITE_END], message, strlen(message));
    }
    close(fd[WRITE_END]);      // MUST DO THIS!
    printf("%d messages sent; sleeping a bit. Please wait...\n", NUM);
    sleep(2);

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

Notes.

- In this program, the read and write calls are not error-checked, which they should be.
- The `read()` is a blocking read by default. It will block waiting for data as long as the write-end of the pipe remains open. Naturally, the process has to write the data into the pipe before it reads it, otherwise it will block forever. If the program does not close the write-end of the pipe before the read-loop starts, it will hang forever, because the read will continue to wait for data. This could be avoided if the read-loop knew in advance exactly how many bytes to expect, but that is pretty pointless.

- Notice that the `read()` calls always read the same amount of data. This example demonstrates that the reader can assemble the data it reads from the pipe 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.

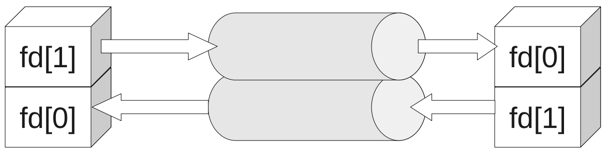
- Finally, observe that before calling `printf()` to print the string, the string has to be null-terminated .

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.

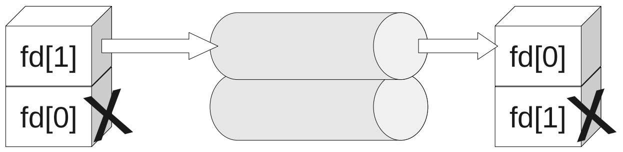
	At one process has the pipe open for writing			**No processes have the pipe open for writing**
	Blocking read		**Non-blocking read**	
Pipe Size (p)	**At least one writer is sleeping**	**No writer is sleeping**		
p = 0	Copy n bytes and return n, waiting for data as necessary when the pipe is empty.	Block 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.			

9.2.1 <u>Parent and Child Sharing a Pipe</u>

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:



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.



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 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.

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

```
#include <stdlib.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(fd[WRITE_FD]);   // MUST DO THIS
        // 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]);    // parent is writing so close read-end
        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 it's 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.

9.2.2 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` ≤ `PIPE_BUF` bytes, the data will not be intermingled. If there is not enough room in the pipe to store `N` ≤ `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.

There are two ways to obtain the value of `PIPE_BUF`. One is simply to include the header file `<limits.h>` and use the macro `PIPE_BUF`, as in

```
#include <limits.h>

char  chunk[PIPE_BUF];
```

A better solution is to obtain the actual value in use in the kernel, in case the header file is not up to date. A call to `fpathconf()` will return the value of various configuration values associated with an open file descriptor:

```
if ( -1 == pipe(fd) )
    exit(1);
long pipe_size = fpathconf(fd[0],  _PC_PIPE_BUF);
```

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. In the above example, the constant `_PC_PIPE_BUF` tells the function to return the value of `PIPE_BUF`. Consult the man page for a complete list of parameters that can be obtained at run-time with this function. POSIX demands that every `PIPE_BUF` be at least 512 bytes. Implementations are free to make it larger. On Linux, `PIPE_BUF` is 4096 bytes.

The following program is designed to demonstrate (but not prove) that writes of up to `PIPE_BUF` bytes are atomic, and that larger writes will not be. 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.

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 and y's will also have a newline. There will be no occurrence of the string Xy or yX in the output because the kernel serializes the concurrent writes.

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

#define  READ_FD  0
#define  WRITE_FD 1
#define  RD_SIZE  10
#define  ATOMIC

int main(int argc, char* argv[] )
{
    int i, repeat;
    int bytesread;
    int mssglen;
    int fd[2];
    int outfd;
    char  mssg[RD_SIZE+1];
    char  *Child1_Chunk, *Child2_Chunk;
    long Chunk_Size;

    if ( argc < 2 ) {
        fprintf(stderr, "Usage:  %s repeats\n", argv[0]);
        exit(1);
    }

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

    repeat     = atoi(argv[1]);
    Chunk_Size = fpathconf(fd[0],  _PC_PIPE_BUF);
#ifndef ATOMIC
    Chunk_Size += 10;   // just make it bigger
```

```
#endif

    Child1_Chunk =  calloc(Chunk_Size, sizeof(char));
    Child2_Chunk =  calloc(Chunk_Size, sizeof(char));
    if ( ( Child1_Chunk == NULL ) || ( Child2_Chunk == NULL ) ) {
        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 ( fork() ) {
    case -1:  // fork failed -- exit
        perror("fork()");
        exit(3);

    case 0:   // child1 code
        mssglen = strlen(Child1_Chunk);
        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 ( fork() ) {
        case -1:  // fork failed -- exit
              perror("fork()");
              exit(5);

        case 0:   // child2 code
             mssglen = strlen(Child2_Chunk);
             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 ( outfd == -1 ) {
                perror("open");
                exit(7);
            }
            close(fd[WRITE_FD]);
            while ((bytesread = read(fd[READ_FD],mssg,RD_SIZE)) != 0)
                if ( bytesread > 0 )
                    write(outfd, mssg, 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);
    }
}
```

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 1000 or more. To check that the results are correct, i.e., that the output is not intermingled when writes are smaller than `PIPE_BUF` bytes each and possibly mingled when larger, first compile it with the `ATOMIC` macro enabled, and then comment it out and run it again. Try the following script each time, with a command line argument of 1000:

```
#!/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
```

You should see output such as this with `ATOMIC` enabled:

```
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
```

and with `ATOMIC` disabled:

```
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 writes are too large, the writes will not be atomic, and a process can be interrupted in the middle of its write.

9.2.3 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, borrowed from [Haviland et al] and modified slightly, can be run on any system to test the maximum size of a pipe.

```
int count;

void on_alarm( int signo)
{
    printf("write() blocked after %d chars\n", count);
    exit(0);
}

int main()
{
    int fd[2];
    int pipe_size;
    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);
    }
    pipe_size = fpathconf(fd[0],  _PC_PIPE_BUF);
    printf("Max size of atomic write is %d bytes\n", pipe_size);

    while (1) {
        alarm(10);
        write(fd[1], &c, 1);
        alarm(0);
        if ( (++count % 1024) == 0 )
            printf ( "%d chars in pipe\n", count);
    }
    return 0;
}
```

Notes.

- Since there is only one process and it will never read 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 expire before the write finishes. When the pipe is full, the alarm will expire and the alarm handler will display the total number of bytes written and then terminate the program.

- The only purpose of displaying the value of `PIPE_BUF` is informational.

9.2.4 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. Some of these have been mentioned already, some not.

- 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 `EPIPE`.

- 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.

- If when a writer is finished using a pipe, it fails to close the write-end of the pipe, and a reader is blocked on a `read()`, the reader will remain permanently blocked; as long as one writer has the pipe open for writing, the `read()` will remain blocked. As soon as all writers close the write-ends of the pipe, the `read()` will return zero.

- A `write()` to a full pipe will block the writer until there are `PIPE_BUF` free bytes in the pipe.

- 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.

- Writes are atomic as long as the number of bytes is smaller than `PIPE_BUF`.

- 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.

- 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.

9.3 I/O Redirection Revisited

9.3.1 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  > list
```

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 `list`.
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 `list`. Then the child `exec()`s "`ls`". The `ls` program thinks it is writing to standard output when it writes to descriptor 1, but it is really writing to the file named `list`. 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 creates a pipe, 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
```

```
// redirectout.c
#include <stdio.h>
#include <unistd.h>
```

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

int main(int argc, char* argv[])
{
    int  pid;
    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:
        close(1);
        fd = open( argv[2], O_WRONLY | O_CREAT | O_TRUNC, 0644 );
        execlp( argv[1],argv[1], NULL );
        perror("execlp");
        exit(1);
    default:
        wait(NULL);
      }
    return 0;
}
```

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.

9.3.2 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. Somehow 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`. 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 duplicate 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.

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

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, `dup()` `fd[0]`, and `exec` the `grep` command. We can put these ideas together in a more general program, called `shpipe1.c`, which follows.

```
//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));          // close stdout
        dup(fd[1]);                     // make stdout point to pipe
        close(fd[0]);                   // close read-end in child
        close(fd[1]);                   // not needed now
        execlp( argv[1],argv[1], NULL );// exec command that writes
        perror("execlp");
        exit(1);
    default:
        close(fileno(stdin));           // close stdin
        dup(fd[0]);                     // make stdin point to pipe
        close(fd[1]);                   // close write-end
        close(fd[0]);                   // not needed now
        execlp( argv[2],argv[2], NULL );// exec command that reads
        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 `fd1` 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.)

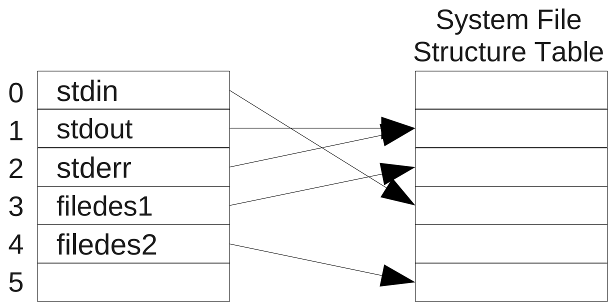


Illustration 1: Initial File Descriptor Table

A picture best illustrates how `dup2()` works. Assume the initial state of the file descriptors for the process is as shown in Illustration 1. Now suppose that the process makes the call

```
dup2( filedes2 , fileno(stdin));
```

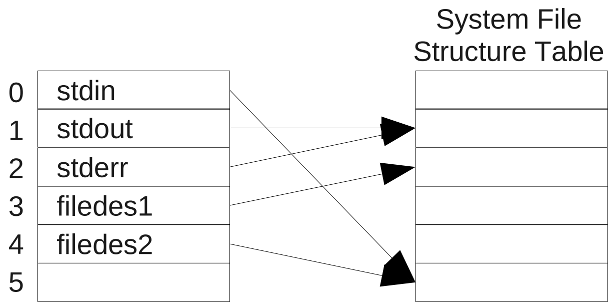


Illustration 2: File Descriptor Table After dup2()

Then, after the call the table is as shown in Illustration 2. Descriptor 0 (standard input) became a copy of `filedes2` as a result of the call. Descriptor `filedes2` is now redundant and can be closed if `stdin` is going to be used instead.

The following program, `shpipe2.c` , is an improved version of `shpipe1.c`.

```
//shpipe2.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:
        dup2(fd[1],fileno(stdout));     // copy fd[1] into stdout
        close(fd[0]);                    // close read-end of pipe
        close(fd[1]);                    // close write-end of pipe
        execlp( argv[1],argv[1], NULL ); // exec command1
        perror("execlp");
        exit(1);
    default:
        dup2( fd[0], fileno(stdin) );    // copy fd[0] into stdin
        close(fd[0]);                    // close read-end of pipe
        close(fd[1]);                    // close write-end of pipe
        execlp( argv[2],argv[2], NULL ); // exec command2
        exit(2);
    }
    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
```

9.3.3 The popen() Library Function

The sequence of (1) generating a pipe, (2) forking a child process, (3) duplicating file descriptors, and (4) executing 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()`. `pclose()` 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.

```
// 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;
}
```

9.4 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 that clients can communicate with, they 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.

9.4.1 Named Pipes at the Command Level

Before we look at how they are created programmatically, 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 first look at how to use `mknod`.

```
$ 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 directory:

```
$ ls -l PIPE

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

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 will terminate as soon as it receives a 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 &
```

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.

```
$ 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.

9.4.2 Programming With Named Pipes

You can read about the `mkfifo` command in the man pages. We turn instead 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. 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 sever.

In this first example, the server creates a public FIFO. The server and client programs know the name of the public FIFO because they will share a common header file that hard-codes the pathname to the file in the file system. Ideally this name would be chosen so that no other processes in the system would ever choose the same file name[3]. This example supports multiple clients.

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.

The server creates the FIFO and opens it for both reading and writing, even if it only needs to read incoming messages on the pipe. This is because the FIFO needs to have at least one process that has it open for writing, otherwise the server will immediately receive an end-of-file on the FIFO and close its reading loop. By opening it up for writing also, the server will simply block on the `read()` to it, waiting for a client to send it data. Since the server never writes to this pipe, it does not matter whether writes are non-blocking or not, but by opening it with the `O_NDELAY` flag, 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 a client tries to write to a FIFO that does not exist, it will fail.

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. The header file is listed first, followed by the client code.

```
// fifo1.h
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <fcntl.h>
#include <string.h>
#include <limits.h>
#include <errno.h>

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

```
// 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);
    }
    // Repeatedly prompt user for command, read it, and send to server
    while (1) {
        memset(text, 0x0, PIPE_BUF);     // zero string
        nbytes = read(fileno(stdin), text, PIPE_BUF);
        if ( !strncmp(QUIT, text, nbytes-1))    // is it quit?
            break;
        write(publicfifo, text, nbytes);
    }
    // User quit; close write-end of public FIFO
    close(publicfifo);
}
```

Comments.

- The client code allows the user to type "quit" to end the program.
- It is not robust. It does not handle any signals and does no clean-up if it is killed by a signal. If the server is not running it will hang and will need to be killed by a signal.

The server code follows.

```
// rcvfifo1.c
#include "fifo1.h"

int main( int argc, char *argv[])
{
    int         nbytes;       // number of bytes read from popen()
    int         n = 0;
    int         dummyfifo;    // file descriptor to write-end of PUBLIC
    int         publicfifo;   // file descriptor to read-end of PUBLIC
    static char buffer[PIPE_BUF];// buffer to store output of command

    // Create public FIFO
    if ( mkfifo(PUBLIC, 0666) < 0 )
        if (errno != EEXIST ) {
            perror(PUBLIC);
            exit(1);
        }
    if ( (publicfifo = open(PUBLIC, O_RDONLY) ) == -1 ||
          ( dummyfifo = open(PUBLIC, O_WRONLY | O_NDELAY )) == -1  ) {
        perror(PUBLIC);
        exit(1);
    }
    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", ++n, 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. It flushes standard output just in case there is no newline in the message.
- It, like the client, does not handle any error conditions other than the pre-existence of the public pipe. If it finds the pipe already exists, it just continues along.

This first example has several shortcomings that will be overcome in the next example.

9.4.3 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 has a special field that contains the name of the private FIFO that the client creates when it starts up. Each message that a client sends has its private FIFO's name. 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. The client will, in turn, open its FIFO for reading. Illustration 3 depicts the relationship between the clients and the server with respect to the shared pipes.

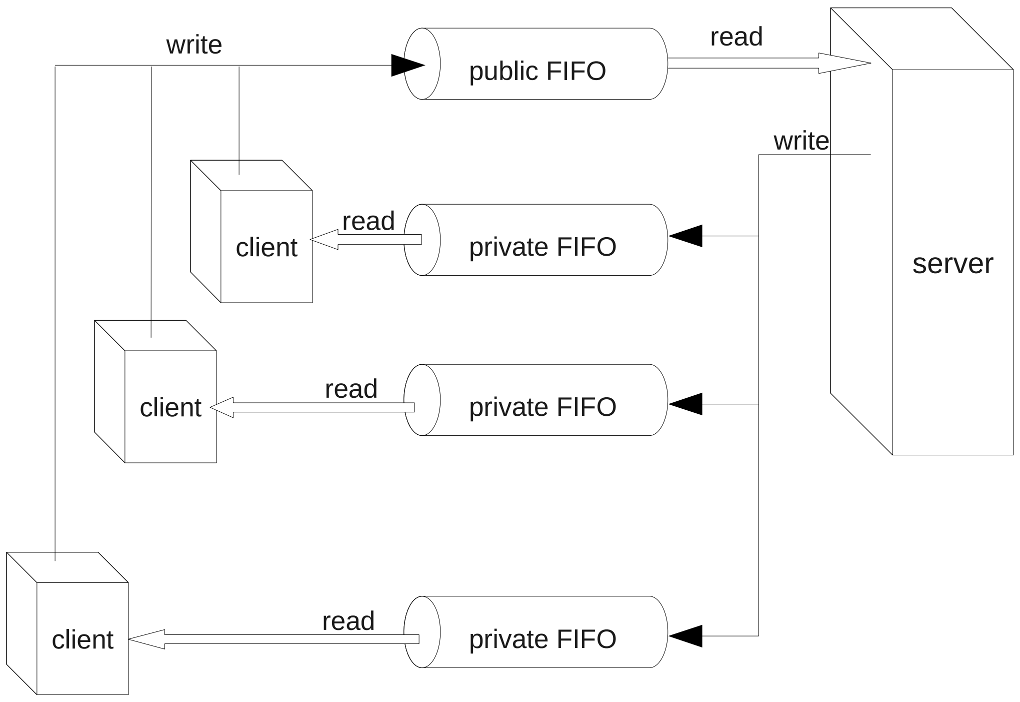


Illustration 3: Client-server Use of FIFOs

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 `upcased` (for uppercase daemon), and the client, `upcaseclient`.

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

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

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

```
struct message {
    char   fifo_name[HALFPIPE_BUF];
    char   text[HALFPIPE_BUF];
};
```

Because the message must be no larger than `PIPE_BUF` bytes, and because it should be flexible enough to allow FIFO pathnames of a large size, the `struct` 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.

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
    i. reads a line from standard input, and
    ii. repeatedly
        a. gets the next `HALFPIPE_BUF-1` sized chunk in the input text,
        b. sends to the server through the public FIFO,
        c. opens its private FIFO for reading,
        d. reads the server's reply from the private FIFO,
        e. copies the server's reply to its standard output, and
        f. 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 code follows.

```
// upcaseclient1.c
#include "upcase1.h"
#define  PROMPT   "string: "
#define  UPCASE   "UPCASE: "
#define  QUIT     "quit"
```

```
const char              startup_msg[] =
     "upcased does not seem to be running. Please start the service.\n";
volatile sig_atomic_t   sig_received = 0;
struct   message        msg;

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;
}

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 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;      /* handler function    */
    if ( sigaction(SIGPIPE, &handler, NULL) == -1 ) {
        perror("sigaction");
        exit(1);
    }

    // Create 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_NDELAY) ) == -1) {
        if ( errno == ENXIO )
            fprintf(stderr,"%s", startup_msg);
        else
            perror(PUBLIC);
        exit(1);

    }

    // Repeatedly prompt user for input, read it, and send to server
    while (1) {
        // Check if a signal was received first, and if so, close
        // write-end of public fifo, remove private fifo and exit
        if ( sig_received ) {
            close(publicfifo);
            unlink(msg.fifo_name);
            exit(0);
        }

        // 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; close write-end of public FIFO and delete private FIFO
    close(publicfifo);
    unlink(msg.fifo_name);
}
```

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 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.
- 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_NDELAY` 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, and 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

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.

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
   i. does a blocking read on the public FIFO,
   ii. 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.)
   iii. converts the message to uppercase,
   iv. writes it to the private FIFO of the client, and
   v. 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.

```
//upcased1.c
#include "upcase1.h"   // fifo.h is shared by sender and receiver

#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);
}

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 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);
    }

    handler.sa_handler = on_sigpipe;      /* handler function    */
    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);
    }
    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 ) {
        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]);

                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 not -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.

9.4.4 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. 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 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
    i. does a blocking `read()` on the public FIFO,
    ii. 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 public FIFO is used exclusively to send "connection" messages to the server. The connection message contains only the information needed to establish the two-way

private communication between the server and the client, namely the names of the two pipes:

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

- 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 client's 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 FIFO, until it received an end-of-file from the client.

The client is also structurally different from the previous client. The major steps it takes are:

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
   i. reads a line from standard input,
   ii. breaks the line into `PIPE_BUF`-sized chunks,
   iii. sends each chunk successively to the server through its raw_text FIFO,
   iv. opens the converted_text FIFO for reading,
   v. reads the converted_text FIFO, and copies its contents to its standard output, and
   vi. closes the read-end of the converted_text FIFO
7. It closes all of its FIFOs and removes the files.

Illustration 4 shows how the client processes and the server parent and child processes use the various FIFOs. Compare this to Illustration 3.

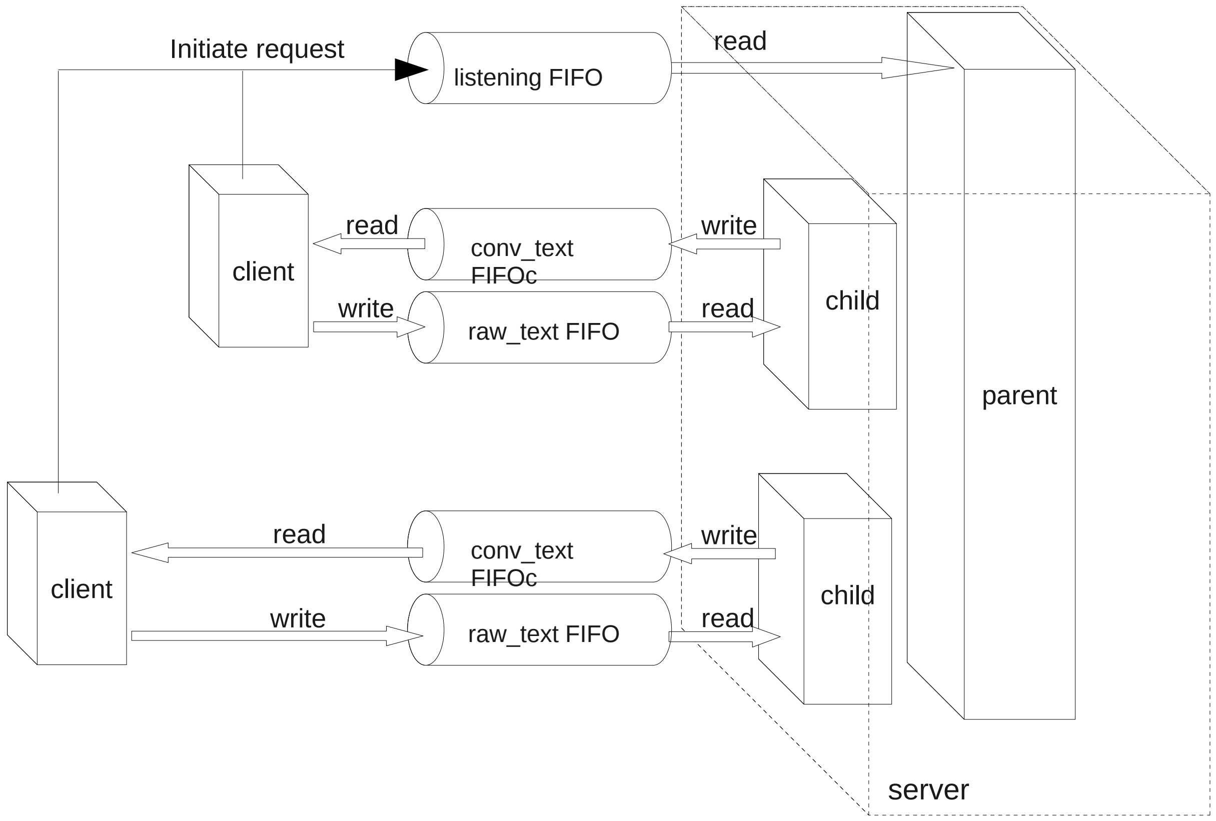


Illustration 4: Use of Pipes in a Concurrent Server

The code for the client is displayed first.

```
//upcaseclient2.c
#define  MAXTRIES 5
const char   server_no_read_msg[] =
"The server is not reading the pipe.\n";

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

const char   missing_pipe_msg[] =
"cannot communicate with the server due to a missing pipe.\n"
"Check if the server is running and restart it if necessary.\n";

int             convertedtext_fd; // client's read FIFO
int             dummyreadfifo;    // to hold write-end open
int             rawtext_fd;       // client's write FIFO
int             publicfifo;       // server's public FIFO
FILE*           input_srcp;       // input stream
struct message msg;

// Signal Handlers
void on_sigpipe( int signo )
{
    fprintf(stderr, "%sExiting...\n",server_no_read_msg);
    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);
}

// Clean up routine
void clean_up()
{
    close(publicfifo);
    close(rawtext_fd);
    unlink(msg.converted_text_fifo);
    unlink(msg.raw_text_fifo);
}

int main( int argc, char *argv[])
{
    int              strLength;
    int              nChunk;
    int              nbytes;
    int              tries = 0;
    static char      buffer[PIPE_BUF];
    static char      textbuf[BUFSIZ];
    struct sigaction handler;

    if ( argc > 1 ) {
        if ( NULL == (input_srcp  = fopen(argv[1], "r")) ) {
            perror(argv[1]);
            exit(1);
        }
    }
    else
        input_srcp = stdin;

    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", noserver_msg);
        else if ( errno == ENOENT )
            fprintf(stderr,"%s %s", argv[0], missing_pipe_msg);
        else {
            fprintf(stderr,"%d: ", errno);
            perror(PUBLIC);
        }
        clean_up();
        exit(1);
    }

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

    // Open the raw text fifo for writing: if server is not reading
    // try a few times and then exit
    while (((rawtext_fd = open(msg.raw_text_fifo,
             O_WRONLY | O_NDELAY )) == -1 ) && (tries < MAXTRIES ))  {
            sleep(1);
            tries++;
    }
    if ( tries == MAXTRIES ) {
        // Failed to open client private FIFO for writing
        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 raw_text 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));
            if ( (convertedtext_fd = open(msg.converted_text_fifo,
                  O_RDONLY) ) == -1) {
                perror(msg.converted_text_fifo);
                clean_up();
                exit(1);
            }

            memset(buffer, 0x0, PIPE_BUF);
            while ((nbytes = read(convertedtext_fd, buffer,PIPE_BUF)) > 0)
                write(fileno(stdout), buffer, nbytes);

            close(convertedtext_fd);
            convertedtext_fd   = -1;
        } // end of for-loop
    }
    clean_up();
}
```

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.

- 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 `unlink()`-ing 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.

The server's code is next.

```
// upcased2.c
#include "upcase2.h"
#include "sys/wait.h"

#define    WARNING  "\nNOTE: SERVER ** NEVER ** accessed private FIFO\n"
#define    MAXTRIES 5
int        dummyfifo;        // file descriptor to write-end of PUBLIC
int        clientreadfifo;   // client's converted_text FIFO
int        clientwritefifo;  // client's raw_text FIFO
int        publicfifo;       // file descriptor to read-end of PUBLIC
FILE*      upcaselog;        // points to log file for server
pid_t      server_pid;       // stores parent pid

void on_sigpipe( int signo );
void on_sigchld( int signo );
void on_signal( int sig );

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

    homepath  = getenv("HOME");
    sprintf(logfilepath, "%s/.upcase_log", homepath );

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

    // Register the signal handlers
    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);
    }

    if ( (publicfifo = open(PUBLIC, O_RDONLY) ) == -1 ||
      ( dummyfifo = open(PUBLIC, O_WRONLY | O_NDELAY )) == -1  ) {
        perror(PUBLIC);
        unlink(PUBLIC);
        exit(1);
    }
    // Store the process id of the parent process for the signal handler
    // to compare to later.
    server_pid = getpid();

    while ( read( publicfifo, (char*) &msg, sizeof(msg)) > 0 ) {
        if ( 0 == fork() ) {
           clientwritefifo = -1;
           if ((clientwritefifo = open(msg.raw_text_fifo,O_RDONLY))==-1) {
               fprintf(stderr, "Client pipe not open for writing\n");
               exit(1);
            }
            // Clear the buffer used for reading the client's text
            memset(buffer, 0x0, PIPE_BUF);
            while ((nbytes = read(clientwritefifo,buffer,PIPE_BUF)) > 0) {
                 // Convert the text to uppercase
                 for ( i = 0; i < nbytes; i++ )
                      if ( islower(buffer[i]))
                          buffer[i] = toupper(buffer[i]);
```

```
                tries = 0;
                // Open client's convertedtext_fd --
                // Try 5 times or until client is reading
                while (((clientreadfifo = open(msg.converted_text_fifo,
                        O_WRONLY|O_NDELAY)) == -1) && (tries < MAXTRIES))
                {
                     sleep(1);
                     tries++;
                }
                if ( tries == MAXTRIES ) {
                    // Failed to open client private FIFO for writing
                    write(fileno(stderr), WARNING, sizeof(WARNING));
                    exit(1);
                }

                // Send converted text to client in its readfifo
                if ( -1 == write(clientreadfifo, buffer, nbytes) ) {
                    if ( errno == EPIPE )
                        exit(1);
                }
                close(clientreadfifo);  // close write-end of private FIFO
                clientreadfifo = -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, "child %d terminated.\n", pid);
        fflush(upcaselog);
    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 this is the parent executing it, remove the public fifo
    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.
- 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.

9.5 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 from Chapter 8 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.

```
#include <syslog.h>
#define  MAXFD  64
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 in the code and insert the line

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

before the first executable statement.

9.6 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[5]. 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>
```

5 There is a similar call named `poll()`.

```
        #include <unistd.h>

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

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[6]. (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` struct 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 1024 in the kernel we are using. 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`:

6 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.

```
      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);
```

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].

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.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] );

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;
}

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 ) {
        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(getpid() % 10 );  // sleep a pseudo random # of seconds
    }
    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 ech 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.

9.7 Sockets

Pipes are a good segue into sockets. ***Sockets*** are used like pipes in many ways but, unlike pipes, they can be used across networks. Sockets allow unrelated processes on different computers on a network to exchange data through a channel, using ordinary `read()` and `write()` system calls. We call this ***remote interprocess communication***. The development of sockets comes from the Berkeley distributions of UNIX in the early 1980's. AT&T developed a different API for remote interprocess communication, called the ***Transport Level Interface*** (TLI) in 1986. In many ways, TLI has advantages over sockets, however, sockets have been around a long time, there is a great deal of code that uses them, sockets can be accessed like files and work the same whether on a local machine or across a network, making programming them easier. TLI programming is more complex; it requires many more structures. This is why we will be looking at sockets here. In order to understand how to use sockets, you need to know the basics of networks.

9.7.1 Connections

There are two ways in which sockets can be used, corresponding roughly to the difference between making a telephone call and having an email conversation with someone. When you make a telephone call to someone, you have a conversation over a dedicated communication channel, the telephone line, and you stay connected with the person on the other end for the duration of the call. In socket parlance this is called a ***connection oriented model***. When you have an email conversation with someone, the messages are sent to the other person across different paths, and there is no dedicated connection. In fact there is no guarantee that the messages that you send will arrive in the order you send them, and the only way for the person who receives them to know who sent them is for them to have a return address in their message header. In socket parlance this is the ***connectionless model***.

The connection oriented model uses the ***Transmission Control Protocol***, known as ***TCP***. The connectionless model uses the ***User Datagram Protocol***, or ***UDP***. There are many important differences between TCP and UDP, or equivalently, between connection oriented and connectionless models, but we cannot go into them at length here. The most important differences are that TCP provides a reliable, full-duplex, sequenced channel with flow control. UDP can be full-duplex but it is not reliable (no guarantee of packet delivery), not sequenced (packets can arrive in different order than they were sent), and has no flow control (a sender can send faster than the receiver can receive).

9.7.2 Communication Basics

In order to understand how to program with sockets, you need to have a basic understanding of the important concepts that underly their use. This includes network addresses, communication domains (not internet domains), protocol families, and socket types.

Network Addresses

For two processes to communicate, they need to know each other's network addresses. At the level of socket programming, a network address consists of two parts: an internet (IP) address and a port number. The IP address, if 32 bits, consists of 4, 8-bit octets, and is expressed in the standard dot-notation as in "`146.95.2.131`", which happens to be the address of eniac. Some computers have multiple network interface cards and therefore may have multiple internet addresses. It used to be the case that internet addresses had a specific structure and were divided into address classes. That is no longer the case. They are now just flat addresses.

The kernel does not represent IP addresses as strings of octets -- that would be inefficient. It uses the `in_addr_t` data type, defined in `<arpa/inet.h>`, to represent an IP address. However, we will see that there are functions to convert from one format to the other.

The server on a machine has to have a specific ***port*** that it uses. There are many analogies that we could use, but if you think of an IP address as specifying a specific company's main telephone line, then the port is like a telephone extension within the company. The server uses a specific port for its services and the clients have to know the port number in order to contact the server. A port is a 16-bit integer.

Certain port numbers are "***well-known***" and reserved by particular applications and services. For example, port 7 is for echo servers, 13 for daytime servers, 22 for SSH, 25 for SMTP, and 80 for HTTP. Port numbers from 1 to 1023 are the well-known ports. To see a list of the port numbers in use, take a look at the file, `/etc/services`.

Ports 1024 through 49151 are registered ports. These numbers are not controlled and a service can use one if it is not already in use.

Ports 49152 through 65536 cannot be used. They are called ***ephemeral ports***, which are assigned automatically by TCP or UDP for client use.

The lsof command can be used to view the ports that are currently open. The command is actually more general than this -- it can be used to view all open files.

To add: The lsof command for seeing ports in use, different definitions from BSD Solaris IANA,

Families

In order for two processes to communicate, they must use the same protocol. Part of the procedure for establishing communication involves specifying the address and protocol family. For example, the family might specify that the protocol is *IPv4* and the addresses are IP addresses, or that the family is *IPv6* with IP addresses, or that addresses are purely local, i.e., on the same machine using UNIX protocols. When a socket is created, the family is specified as one of the arguments to the function.

Socket Types

When a socket is created, its type must be specified. The type corresponds to the type of connection. A connection oriented communication has type `SOCK_STREAM`, whereas a connectionless communication is of type `SOCK_DGRAM`. There are also raw sockets, with type `SOCK_RAW`.

9.7.3 The Socket Interface

What then is a socket? A socket is an *endpoint of a two-way communication channel.* The `socket()` system call creates this endpoint and returns a file descriptor that represents it. We do not have to know how a socket is implemented to use it, however, you should think of a socket as something like the file structure that represents a file in UNIX. It is an internal structure in the kernel, accessed through a file descriptor, representing one end of a communication channel that has a specific network address, family (also called domain), port number, and socket type.

The `socket()` function creates what is called an unnamed socket:

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

int socket( int domain, int type, int protocol);
```

The domain is an integer specifying the address family and protocol. These families are defined in `<sys/socket.h>`. Some of the common domain values are

Name	*Purpose*
`PF_UNIX`	Local communication
`PF_INET`	IPv4 Internet protocols
`PF_INET6`	IPv6 Internet protocols

The `type` can be one of `SOCK_STREAM`, `SOCK_DGRAM`, or `SOCK_RAW` or several others. The `SOCK_STREAM` type is the connection model, with full-duplex, reliable, sequenced transmissions.

The `protocol` can be used to specify a particular protocol in the case that there is more than one choice for the particular type of socket and address family. Setting it to 0 ensures that the kernel will pick the appropriate protocol.

The return value of `socket()` is a file descriptor that can be used to read or write the socket. As an example,

```
sockfd = socket(PF_INET, SOCK_STREAM, 0);
```

creates a connection-oriented socket that can be used for communication over the internet.

9.7.4 Setting Up a Connection Oriented Service

We will go through the steps that a server must take in a connection-oriented model. The basic steps that a server must take are below. Details on how to use the specific functions will follow.

1. Create a socket using `socket()`. This creates an endpoint of communication, but does not associate any particular internet address or port number to it.

2. Bind the socket to a local protocol address using `bind()`. This gives a "name" to the socket.

3. The socket created so far is an active socket, one that can connect to other sockets actively, like dialing another telephone number. Since this is the server, the purpose of this socket is not to "dial-out" but to listen for incoming calls. Therefore, the server must now call `listen()` to tell the kernel that all it really wants to do is listen for incoming messages and set a limit on its queue size. This call will basically put the socket into the LISTEN state in the TCP protocol.

   After `listen()` has returned, two queues have been created for the server. One queue stores incoming connection requests that have not yet completed the TCP handshake protocol. The other queue stores incoming requests that have completed the handshake. These requests are ready to be serviced.

4. Enter a loop in which it repeatedly accepts new connections and processes them. It can accept a new connection with the `accept()` call. The `accept()` function removes the request at the front of the completed connection queue and creates a second socket that the server can use for talking with this client. The return value of `accept()` is a file descriptor that represents this socket. The original socket continues to exist. The idea is that the original socket is just for listening, not talking to clients. In fact it is called the ***listening socket***, and the new socket is called the ***connected socket***. When the connection is closed, this connected socket is removed.

That is the essence of the server's tasks. Now what remains is to see how to program this.

9.7.5 Programming a Connection Oriented Server

We have already seen how the `socket()` call works. The step of binding a local protocol address to the socket is carried out with `bind()`, but before we look at `bind()` we need to see how these addresses are represented. A ***generic socket*** address is defined by the `sockaddr struct` defined in `<sys/socket.h>`:

```
struct sockaddr {
    sa_family_t sa_family;  /* address family */
    char        sa_data[];  /* socket  address */
};
```

This is a generic socket address structure because it is not specific to any one address family. When you call `bind()`, you will be specifying a particular family, such as `PF_INET` or `PF_UNIX`. For each of these there is a different form of socket address structure. The address structure for `PF_INET`, defined in `<netinet/in.h>`, would be

```
struct sockaddr_in {
    sa_family_t    sin_family;  /* internet address family */
    in_port_t      sin_port;    /* port number */
    struct in_addr sin_addr;    /* IP  address */
    unsigned char  sin_zero[8]; /* padding  */
};
```

The `bind()` system call takes the socket file descriptor returned by `socket()` and an address structure like the one above, and "binds" them together to form the end of a socket that can now be used by processes out there in internet land to find this server:

```
#include <sys/types.h>
#include <sys/socket.h>
int bind(int sockfd, const struct sockaddr *address,
         socklen_t addrlen);
```

Putting these few steps together, we might start out with the following code:

```
int listenfd;
int size  = sizeof(struct sockaddr_in);

struct sockaddr_in server = {PF_INET, 25555, INADDR_ANY};

if ( (listenfd = socket(PF_INET, SOCK_STREAM, 0)) == -1 )
{
    perror("socket call failed");
    exit(1);
}

if ( bind( listenfd, ( struct sockaddr *) &server, size ) == -1 )
{
    perror(" bind call failed");
    exit(1);
}
```

The `sockaddr_in struct` is initialized to use the IPv4 protocol family with a port of `25555`, large enough to be safe for our purposes, and `INADDR_ANY` as the local IP address. Specifying this constant means that if there is more than one IP address for this host, any will do. The bind call is given the `listenfd` descriptor, the address of this struct, and its `size`.

The next step is to call `listen()`, which is defined by

```
#include <sys/socket.h>
int listen(int sockfd, int queue_size);
```

The first argument is the descriptor for the already bound socket, and the second is the maximum size of the queue of pending (incomplete) connections.

The `accept()` call is defined as follows:

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

int accept(int sockfd, struct sockaddr *addr,
           socklen_t *addrlen);
```

The `accept()` call expects the socket descriptor of a socket that has been created with `socket()`, bound to a local address with `bind()`, and set to listen with `listen()`. The second argument, if not `NULL`, is a pointer to a generic socket address structure, and the third is the address of a variable that stores its length in bytes. After the call, the address will be filled with the client's socket address, and the size will reflect the true size of the client's specific socket address `struct`. The return value will be the descriptor of a connected socket. The `accept()` will block waiting for a connection.

We can put all of this together in a simple concurrent server that, yes, once again, does lower to upper case conversion. This time it will handle just one character at a time. We will move on to a more interesting task afterwards. This code is based on an example from [Haviland et al]. It forks a child process to handle each incoming connection. The client and the server will share a common header file, `sockdemo1.h`, which is displayed first, followed by the server code.

```
// sockdemo1.h
#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>
#include <ctype.h>
#include <signal.h>
#include <errno.h>

#define SOCKADDR       (struct sockaddr*)
#define SIZE           sizeof(struct sockaddr_in)
#define DEFAULT_HOST   "eniac.geo.hunter.cuny.edu"
#define PORT           25555
#define ERROR_EXIT( _mssg, _num)  perror(_mssg);exit(_num);
```

Comments.

- (Perhaps out of laziness, I finally wrote a little macro (`ERROR_EXIT`) so that I do not have to keep typing the `perror(); exit()` combination on failures of system calls. It is included in this header file. Rather than making it a function, I made it a macro so that the code is faster.)
- The `SOCKADDR` macro reduces typing.
- The server and client are compiled with the same header so that the port number is hard-coded into each. The number 25555 appears to be unused on all of the machines I have run this example on.

The server code follows.

```
//sockdemo1_server.c
#include "sockdemo1.h"
#define LISTEN_QUEUE_SIZE   5

// The following typedef simplifies the function definition after it
typedef void    Sigfunc(int);   /* for signal handlers */

// override existing signal function to handle non-BSD systems
```

```
Sigfunc*  Signal(int signo, Sigfunc *func);

// Signal handlers
void on_sigpipe(int sig);
void on_sigchld(int sig);

// This needs to be global because the signal handler has to access it
int connectionfd;

int main(int argc, char* argv[])
{
    int listenfd;    // holds the file descriptor for the socket
    char c;
    struct sockaddr_in server = {PF_INET, PORT, INADDR_ANY};

    Signal(SIGCHLD, on_sigchld);
    Signal(SIGPIPE, on_sigpipe);

    if ( (listenfd = socket(PF_INET, SOCK_STREAM, 0)) == -1 ) {
        ERROR_EXIT("socket call failed",1)
    }

    if ( bind( listenfd, ( struct sockaddr *) &server, SIZE ) == -1 ) {
        ERROR_EXIT(" bind call failed",1)
    }

    if (listen(listenfd, LISTEN_QUEUE_SIZE ) == -1) {
        ERROR_EXIT(" listen call failed",1)
    }

    for ( ; ; ) {
        if ( (connectionfd = accept(listenfd, NULL, NULL ) ) == -1 ){
            if (errno == EINTR )
                continue;
            else
                perror(" accept call failed");
        }
        switch ( fork() ) {
        case -1:
            ERROR_EXIT("fork call failed",1)
        case  0:
            while ( recv(connectionfd, &c, 1, 0) > 0 ) {
                c = toupper(c);               // convert c to upeprcase
                send(connectionfd, &c, 1, 0 );// send it back
            }
            close(connectionfd);
            exit(0);
        default:
            // server code
            close(connectionfd);
        }
    }
}

void on_sigpipe( int sig)
{
    close(connectionfd);
    exit(0);
}

void on_sigchld(int sig)
{
    pid_t   pid;
    int     stat;

    pid  = wait(NULL);
    return;
}

Sigfunc*  Signal(int signo, Sigfunc *func)
{
    struct  sigaction act, oact;

    act.sa_handler = func;
    sigemptyset(&act.sa_mask);
    act.sa_flags  = 0;
    if (SIGALRM != signo ) {
        act.sa_flags |= SA_RESTART;
    }
    if ( sigaction(signo, &act, &oact) < 0 )
        return ( SIG_ERR );
    return ( oact.sa_handler);
}
```

Comments.

- This program uses a user-defined `Signal()` function to encapsulate the logic of registering the signal handlers. Since we have been registering multiple handlers in most of our programs, it would have been a good idea to create this function earlier and put it in a library to reuse. The idea for this is from [Stevens].

- The program could have used ordinary `read()` and `write()` system calls. Instead, as a way to introduce two socket-specific communications primitives, it uses `recv()` and `send()`. `recv()` is one of a set of three socket-reading functions:

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

  ssize_t recv(int s, void *buf, size_t len, int flags);
  ssize_t recvfrom(int s, void *buf, size_t len, int flags,
                   struct sockaddr *from, socklen_t *fromlen);

  ssize_t recvmsg(int s, struct msghdr *msg, int flags);
  ```

- The `recvfrom()` and `recvmcg()` functions are the most general. The prototype for `recv()` is the same as that of `read()` except that it has a fourth argument that can be used to set various flags to control how the `recv()` behaves. The flags can be used to turn on non-blocking operation (`MSG_DONTWAIT`), to notify the kernel that the process wants to receive out-of-band data (`MSG_OOB`), or to peek at the data without reading it (`MSG_PEEK`), to name a few. In our program, no flags are used, so the fourth argument is zero, and `recv(s, buf, n, 0)` is identical to `read(s, buf, n)`.

- The `send()` function is also one of a set of three:

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

  ssize_t send(int s, const void *buf, size_t len, int flags);
  ssize_t sendto(int s, const void *buf, size_t len, int flags,
              const struct sockaddr *to, socklen_t tolen);
  ssize_t sendmsg(int s, const struct msghdr *msg, int flags);
  ```

- The `sendto()` and `sendmsg()` functions are the most general. The prototype of send is identical to that of `write()` except for the additional argument. Like `recv()`, the fourth argument of `send()` is a set of flags that can be or-ed together. Some of the flags are the same, such as `MSG_OOB`, which allows the process to send out-of-band data. See the man page for more details. We will return to the use of `sendto()` and `recvfrom()` when we look at a connection-less server.

The code for the client is next.

```
// sockclient1.c
#include "sockdemo1.h"

int main(int argc, char** argv)
{
```

```
    int                 sockfd;
    char                c, rc;
    char                ip_name[256] = "";
    struct sockaddr_in server;
    struct hostent     *host;

    if ( argc < 2 )
       strcpy(ip_name, DEFAULT_HOST);
    else
       strcpy(ip_name, argv[1]);

    if ( (host = gethostbyname(ip_name)) ==  NULL) {
        ERROR_EXIT("gethostbyname", 1);
    }

    memset(&server, 0, sizeof(server));
    memcpy(&server.sin_addr, SOCKADDR *host->h_addr_list, SIZE);
    server.sin_family = AF_INET;
    server.sin_port   = PORT;

    if ( (sockfd = socket(AF_INET, SOCK_STREAM, 0) ) == -1 ) {
        ERROR_EXIT("socketcall failed",1)
    }

    if ( connect (sockfd, SOCKADDR &server, sizeof(server)) == -1) {
        ERROR_EXIT("connect call failed",1);
    }

    for ( rc = '\n';;) {
        if (rc == '\n')
            printf("Input a lowercase character\n");
        c = getchar();
        write(sockfd, &c, 1);
        if ( read(sockfd, &rc, 1) > 0 )
            printf("%c", rc);
        else {
            printf("server has died\n");
            close(sockfd);
            exit(1);
        }
    }
}
```

Comments.

- The client does hostname-to-address translation to make it more generic. The user can supply the name of the server on the command line rather than having to remember the IP address. If the IP address changes, the program still works. The `gethostbyname()` call returns a pointer to a `hostent struct`, given a string that contains a valid hostname.

```
#include <netdb.h>
extern int h_errno;

struct hostent *gethostbyname(const char *name);
```

- The `hostent struct` is defined in the `<netdb.h>` header file:

```
struct hostent {
    char  *h_name;            /* official name of host */
    char **h_aliases;         /* alias list */
    int    h_addrtype;        /* host address type */
    int    h_length;          /* length of address */
    char **h_addr_list;       /* list of addresses */
}
#define h_addr h_addr_list[0] /* for backward compatibility */
```

- The `h_name` field is the official name of the host. For example, if it is given the name "`eniac`" when running on a host on our network, it is able to resolve the name, and the `h_name` field will be filled in with "`eniac.geo.hunter.cuny.edu`". The `aliases` member is a pointer to a list of strings, each of which is an alias, i.e., another name listed in the hosts database for the same machine. The `h_addr_list` is a a pointer to a list of internet addresses for this host. If the host has just one network interface card, then only `h_addr_list[0]` is defined. Each entry is of type `in_addr`, which is why, in the client code, if can be assigned directly (with a cast) to the `sin_addr` field of the `sock_addr struct`. We do not use the `h_addrtype` or `h_length` fields here.

- The client uses ordinary `read()` and `write()` calls for its I/O operations.

9.7.6 A Connection-Oriented Client Using Multiplexed I/O

Consider the client from the `upcase` example in Section 9.4.4. It reads a line from standard input, writes it into a pipe, and then reads the converted text from a second pipe. In that example, two pipes were needed because a pipe cannot be used as a bi-directional channel. However, we can replace the pair of pipes by a single socket, which can then be used for both sending the raw text to the server and receiving the converted text from it. In addition, by making the socket an Internet-domain socket, the client and server can be on different machines.

If we keep the original design for the client but just replace the pipes by a socket, the client would read the raw text from standard input, write it to the socket, and then read the converted text from the same socket, in a loop of the form

```
    while ( true ) {
        get text from standard input;
```

```
          write text to socket;
          read converted text from socket;
          write response on standard output;
      }
```

Since input can be redirected, it can arrive much faster than the responses that it receives from the server, because the server might be a long distance away. The client would spend most of its time blocked on the call to read the socket, even though both the server and the socket itself could handle much larger throughput. The same thing could happen in the interactive case as well if the user enters text very quickly but the round-trip time for the socket is large. In this case the client would be delayed in displaying a prompt to the user on the terminal. Therefore, it makes sense in this client to multiplex the standard input and the socket input using the `select()` call. By using the `select()` call, the client will only block if neither the user nor the server has data to read. As long as text arrives on the standard input stream, it will be forwarded to the server. If text arrives on standard input much faster than the round-trip time, the text will keep being sent to the server, which will process the lines one after the other and send them back in a steady stream.

An analogy will help. Imagine a thirty-person fire brigade trying to put out a fire with a single bucket. The bucket is filled with water and passed from one person to the next to the fire, poured on the fire, and then passed back to the water supply, where this is repeated. Suppose it takes one minute for the round trip and the bucket holds 5 gallons of water. This supplies 5 gallons per minute to the fire. Now suppose there are sixty buckets available. The first bucket is filled and handed to the next person, and the second bucket is filled, and so on, until all sixty buckets are filled. Assuming the people know how to pass the full buckets past the empty buckets and the exchange rate is uniform, although the round-trip time has not changed, there will be sixty buckets in the brigade at each instant, and each second, a full bucket will arrive at the fire. The fire will be supplied 5 gallons per second, or 300 gallons per minute.

This is how using `select()` can increase the throughput in the case that the bottleneck is the length of time it takes for the data to make a round trip from client to server and back. The code follows. It uses the same header file as was used in `sockdemo1`. This client will not accept a file name on the command line; it uses a single command line argument, which is the name of the host on which the server is running.

```
//sockdemo2_client.c
#include "sockdemo1.h"

#define  MAXFD( _x, _y)  ((_x)>(_y)?(_x):(_y))

int main(int argc, char** argv)
{
    int                sockfd;
    char               c, rc;
    char               ip_name[256] = "";
    struct sockaddr_in server;
    struct hostent     *host;
    fd_set              readset, copy;
    int                 maxfd, n;
    char                recvline[MAXLINE];
    char                sendline[MAXLINE];
    int                 stdin_eof = 0;

    if ( argc < 2 )
       strcpy(ip_name, DEFAULT_HOST);
    else
       strcpy(ip_name, argv[1]);

    if ( (host = gethostbyname(ip_name)) ==  NULL) {
        ERROR_EXIT("gethostbyname", 1);
    }

    memset(&server, 0, sizeof(server));
    memcpy(&server.sin_addr, SOCKADDR *host->h_addr_list, SIZE);
    server.sin_family = AF_INET;
    server.sin_port   = PORT;

    if ( (sockfd = socket(AF_INET, SOCK_STREAM, 0) ) == -1 ) {
        ERROR_EXIT("socketcall failed",1)
    }

    if ( connect (sockfd, SOCKADDR &server, sizeof(server)) == -1) {
        ERROR_EXIT("connect call failed",1);
    }

    maxfd = MAXFD(fileno(stdin), sockfd) +1;

    while ( 1 ) {
        FD_ZERO(&readset);
        if ( stdin_eof == 0 )
            FD_SET(fileno(stdin), &readset);
        FD_SET(sockfd, &readset);
        if ( select( maxfd, &readset, NULL, NULL, NULL ) > 0 ) {
            if ( FD_ISSET(sockfd, &readset)) {
                if ( ( n = read(sockfd, recvline, MAXLINE-1)) == 0 ) {
                     if (stdin_eof == 1)
                         return;
                     else
                         ERROR_EXIT("Server terminated prematurely.", 1);
                }
                recvline[n] = '\0';
                fputs(recvline, stdout);
            }
            if ( FD_ISSET(fileno(stdin), &readset)) {
                if ( fgets(sendline, MAXLINE-1, stdin) == NULL ) {
                    stdin_eof = 1;
                    shutdown(sockfd, SHUT_WR);
                    FD_CLR(fileno(stdin), &readset);
                    continue;
                }
                write(sockfd, sendline, strlen(sendline));
            }
        }
    }
}
```

Comments.

- This client sends entire lines, one after the other, to the server. It appends a null character to each line it receives before printing it to standard output, even though in principle all lines received should be null-terminated, since they are identical to the null-terminated lines that it sent to the server, except for conversion of lowercase to uppercase letters in the line.

- The `shutdown(sockfd, SHUT_WR)` system call turns off writing to the socket. When the client detects the end-of-file on the standard input stream, it cannot close the socket completely, because if it did, it would not receive any lines sent to the server but not yet converted to uppercase. On the other hand, it has to send a notification to the server that there is no more input on the socket, so that the server's `read()` on the socket can return. The `shutdown()` accomplishes this; the server's read() returns and the socket stays open until the server closes its end of the socket. Without `shutdown()` there would be no way to achieve this. Its synopsis is:

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

  int shutdown(int s, int how);
  ```

  Here, the integer `how` can be replaced by one of `SHUT_WR`, `SHUT_RD`, or `SHUT_RDWR`.

- Once the client detects the end-of-file, it also sets a flag for itself, `stdin_eof`, which it uses to decide whether to set a bit in the descriptor mask for the standard input. If end-of-file has been detected, it stops setting that bit; otherwise it sets it. In addition, when the `read()` on the socket returns 0 bytes, it uses this flag to distinguish between two cases: whether the server has stopped sending text because there is none left to send, or there was an error on the socket before the end-of-file condition occurred.

The server's main function is displayed below. Because the signal handling code is no different in this example than in `sockdemo1_server`, it is not included.

```
//sockdemo2_server.c
#include "sockdemo1.h"
#include "sys/wait.h"

#define LISTEN_QUEUE_SIZE   5
typedef void    Sigfunc(int);   /* for signal handlers */

Sigfunc*  Signal(int signo, Sigfunc *func);
void on_sigchld( int signo );
void convert(int sockfd);


int main(int argc, char **argv)
{
    int                 listenfd, connfd;
    pid_t               childpid;
    socklen_t           clilen;
    struct sockaddr_in  clientaddr, server_addr;

    if ( (listenfd = socket(AF_INET, SOCK_STREAM, 0)) == -1 ) {
        ERROR_EXIT("socket call failed",1)
    }

    memset(&server_addr, 0, sizeof(server_addr));
    server_addr.sin_family      = AF_INET;
    server_addr.sin_addr.s_addr = htonl(INADDR_ANY);
    server_addr.sin_port        = PORT;

    if ( bind(listenfd,SOCKADDR &server_addr,sizeof(server_addr)) == -1){
        ERROR_EXIT(" bind call failed",1)
    }
    if (listen(listenfd, LISTEN_QUEUE_SIZE ) == -1) {
        ERROR_EXIT(" listen call failed",1)
    }
    Signal(SIGCHLD, on_sigchld);

    while ( 1 ) {
        clilen = sizeof(clientaddr);
        if ((connfd = accept(listenfd,SOCKADDR &clientaddr,&clilen))<0) {
            if (errno == EINTR)
                continue;
            else
                ERROR_EXIT("accept error",1);
        }
        if ( (childpid = fork()) == 0) {
            close(listenfd);
            convert(connfd);
            exit(0);
        }
        close(connfd);
      }
}

void convert(int sockfd)
{
    ssize_t      n;
    int         i;
    char         line[MAXLINE];

    while ( 1 ) {
        if ( (n = read(sockfd, line, MAXLINE-1)) == 0)
            return;

    for ( i = 0; i < n; i++ )
        if ( islower(line[i]))
            line[i] = toupper(line[i]);
            write(sockfd, line, n);
    }
}
```

Comments.

- All of the logic is encapsulated in the `convert()` function, which the child executes. `convert()` reads the connected socket until it receives the end-of-file and then it terminates, which causes the child to exit in main.

9.8 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. Sockets are a way for processes on the same host or different hosts to exchange data. Unlike pipes, sockets are bi-directional.

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.

Other methods of interprocess communication not discussed in this chapter are shared memory, memory-mapped files, and message queues.

References

[Haviland et al] Haviland, Keith, Dina Gray, & Ben Salama. Unix System Programming. Addison-Wesley. 1999.

[Stevens] Stevens, Richard W.. UNIX Network Programming, Volume 1. Prentice Hall. 1998.

Alphabetical Index