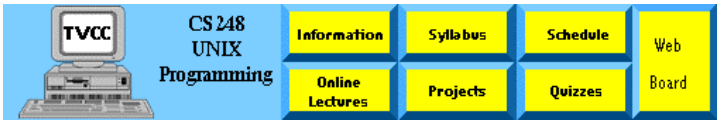


UNIX Programming

"Chapter Eleven - Inter-process Communication: Pipes"

Chapter Outline

Lecture Notes

Inter-process Communication: Pipes

Now, we look at pipes which allow more useful data to be exchanged between processes.

Here are some of the things you need to understand.

- The definition of a pipe
- Process pipes
- Pipe calls
- Parent and child processes
- Named pipes: FIFOs
- Client/server considerations

What is a Pipe?

We use the word **pipe** when we connect a data flow from one process to another.

Shell commands can be linked together so that the output of one process is fed straight to the input of another.

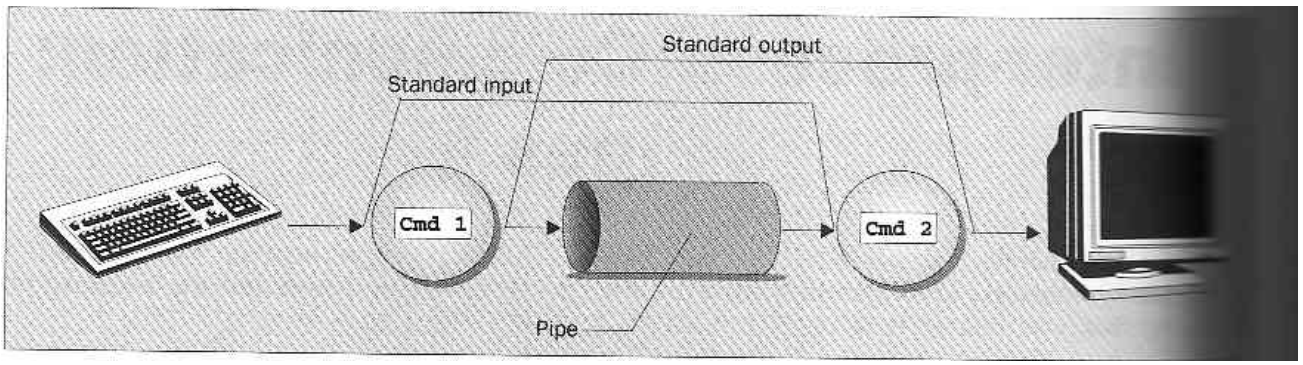
For shell commands, this is entered as:

```
cmd1 | cmd2
```

The shell arranges the standard input and output of the two commands, so that:

- The standard input to **cmd1** comes from the terminal keyboard.
- The standard output from **cmd1** is fed to **cmd2** as its standard input.
- The standard output from **cmd2** is connected to the terminal screen.

The shell has reconnected the standard input and output streams so that data flows from the keyboard input through the two commands and is then output to the screen.



Process Pipes

Perhaps the simplest way of passing data between two programs is with the **popen** and **pclose** functions. These have the prototypes:

```
#include <stdio.h>

FILE *popen(const char *command, const char *open_mode);
int pclose(FILE *stream_to_close);
```

popen

The **popen** function allows a program to invoke another program as a new process and either pass data to or receive data from it.

pclose

When the process started with **popen** has finished, we can close the file stream associated with it using **pclose**.

Try It Out - Using popen and pclose

Having initialized the program, we open the pipe to **uname**, making it readable and setting **read_fp** to point to the output.

At the end, the pipe pointed to by **read_fp** is closed

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

int main()
{
    FILE *read_fp;
    char buffer[BUFSIZ + 1];
    int chars_read;
    memset(buffer, '\0', sizeof(buffer));
    read_fp = popen("uname -a", "r");
    if (read_fp != NULL) {
        chars_read = fread(buffer, sizeof(char), BUFSIZ, read_fp);
        if (chars_read > 0) {
            printf("Output was:-\n%s\n", buffer);
        }
        pclose(read_fp);
        exit(EXIT_SUCCESS);
    }
    exit(EXIT_FAILURE);
}
```

When we run this program on one of the author's machine, we get:

```
$ popen1
Output was:-
Linux stones 1.2.8 #1 Mon Sep 18 18:20:08 BST 1995 i586
```

How It Works

The program uses the **popen** call to invoke the **uname** command. It read some information and prints it to the screen.

Sending Output to popen

Here's a program, **popen2.c**, that pipes dta to another. Here, we use the **od** (octal dump).

Try It Out -Sending Output to an External Program

Have a look at the following code, even type it in if you like...

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

int main()
{
    FILE *write_fp;
    char buffer[BUFSIZ + 1];

    sprintf(buffer, "Once upon a time, there was...\n");
    write_fp = popen("od -c", "w");
    if (write_fp != NULL) {
        fwrite(buffer, sizeof(char), strlen(buffer), write_fp);
        pclose(write_fp);
        exit(EXIT_SUCCESS);
    }
    exit(EXIT_FAILURE);
}
```

When we run this program, we get the output:

```
$ popen2
0000000   O   n   c   e       u   p   o   n       a       t   i   m   e
0000020   ,       t   h   e   r   e       w   a   s   .   .   .  \n
0000037
```

How It Works

The program uses **popen** with the parameter **w** to start the **od -c** command, so that it can send data to it. The results are printed.

From the command line, we can get the same output with the command:

```
$ echo "Once upon a time, there was..." | od -c
```

Passing More Data

Multiple **fread** and **fwrite** can be used to process more data.

Try It Out - Reading Larger Amounts of Data from a Pipe

Here's a program, **popen3.c**, that reads all of the data from a pipe by using multiple **fread**.

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

int main()
{
    FILE *read_fp;
    char buffer[BUFSIZ + 1];
    int chars_read;

    memset(buffer, '\0', sizeof(buffer));
    read_fp = popen("ps -ax", "r");
    if (read_fp != NULL) {
        chars_read = fread(buffer, sizeof(char), BUFSIZ, read_fp);
        while (chars_read > 0) {
            printf("Reading:-\n %s\n", buffer);
            chars_read = fread(buffer, sizeof(char), BUFSIZ, read_fp);
        }
        pclose(read_fp);
        exit(EXIT_SUCCESS);
    }
    exit(EXIT_FAILURE);
}
```

The output we get, edited for brevity, is:

```
$ popen3
Reading:-
   PID TTY STAT  TIME COMMAND
     1  ?  S     0:00 init
     6  ?  S     0:00 bdflush (daemon)
     7  ?  S     0:00 update (bdflush)
    24  ?  S     0:00 /usr/sbin/crond -110
    39  ?  S     0:00 /usr/sbin/syslogd
...
   240 v02 S     0:02 emacs draft1.txt
Reading:-
   368 v04 S     0:00 popen3
   369 v04 R     0:00 ps -ax
...
```

How It Works

The progran uses **popen** with an **r** parameter, so it continues reading from the file stream until there is no more data available.

How popen is Implemented

The **popen** call runs the program you requested by first invoking the shell, **sh**, passing it the **command** string as an argument.

This has two effects, one good, the other not so good.

1. invoking the shell allows complex shell commands to be started with **popen**.

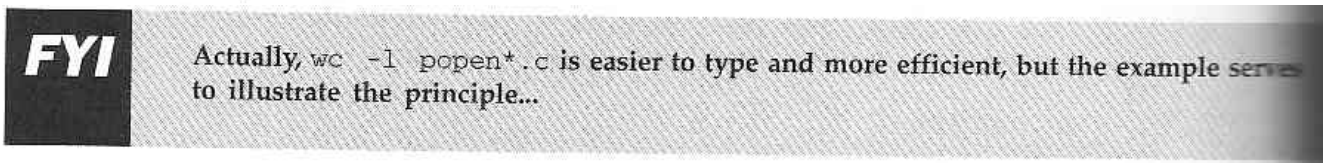
2. Each call to **popen** invokes the requested program and the shell program. So, each call to **popen** then results in two extra processes being started.

We can count all the lines in example program by **cat**ing

the files and then piping its output to **wc -1** , which counts the number of lines.

On the command line, we would use:

```
$ cat popen*.c | wc -l
```



Try It Out - popen Starts a Shell

This program uses exactly the command given above, but through **popen** so that it can read the results:.

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

int main()
{
    FILE *read_fp;
    char buffer[BUFSIZ + 1];
    int chars_read;

    memset(buffer, '\0', sizeof(buffer));
    read_fp = popen("cat popen*.c | wc -l", "r");
    if (read_fp != NULL) {
        chars_read = fread(buffer, sizeof(char), BUFSIZ, read_fp);
        while (chars_read > 0) {
            printf("Reading:-\n %s\n", buffer);
            chars_read = fread(buffer, sizeof(char), BUFSIZ, read_fp);
        }
        pclose(read_fp);
        exit(EXIT_SUCCESS);
    }
    exit(EXIT_FAILURE);
}
```

when we run this program, the output is:

```
$ popen4
Reading:-
      101
```

How It Works

The program shows that the shell is bing invoked to expand **popen*.c** to the list of all files starting with **popen** and ending in **.c** and also feed the output from **cat** into **wc**.

The Pipe Call

The **pipe** function has the prototype:

```
#include <unistd.h>

int pipe(int file_descriptor[2]);
```

pipe is passed an array of two integer file descriptors. It fills the array with two new file descriptors and returns a zero.

Some errors defined in the Linux **man** pages for the operation are:

EMFILE	Too many file descriptors are in use by the process.
ENFILE	The system file table is full.
EFAULT	The file descriptor is not valid.

Any data written to **fijle_descriptor[1]** can be read back from **file_descriptor[0]**.

It's important to realize that these are file descriptors, not file streams, so we must use the lower-level ***read*** *and* ***write*** *calls to access the data, rather than* ***fread*** *and* ***fwrite****.*

Try It Out - The pipe Function

Here's a program, **pipe1.c**, that uses **pipe** to create a pipe..

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

int main()
{
    int data_processed;
    int file_pipes[2];
    const char some_data[] = "123";
    char buffer[BUFSIZ + 1];

    memset(buffer, '\0', sizeof(buffer));

    if (pipe(file_pipes) == 0) {
        data_processed = write(file_pipes[1], some_data, strlen(some_data));
        printf("Wrote %d bytes\n", data_processed);
        data_processed = read(file_pipes[0], buffer, BUFSIZ);
        printf("Read %d bytes: %s\n", data_processed, buffer);
        exit(EXIT_SUCCESS);
    }
    exit(EXIT_FAILURE);
}
```

When we run this program, the output is:

```
$ pipe1
Wrote 3 bytes
Read 3 bytes: 123
```

How It Works

The program creates a **pipe** using the two file descriptors **file_pipes[]**. It then writes data into the pipe using the file descriptor **file_pipes[1]** and reads it back from **file_pipes[0]**.

Try It Out - Pipes across a fork

1. This is **pipe2.c**. It start rather like the first examples, up until we make the call to **fork**.

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

int main()
{
    int data_processed;
    int file_pipes[2];
    const char some_data[] = "123";
    char buffer[BUFSIZ + 1];
    int fork_result;

    memset(buffer, '\0', sizeof(buffer));

    if (pipe(file_pipes) == 0) {
        fork_result = fork();
        if (fork_result == -1) {
            fprintf(stderr, "Fork failure");
            exit(EXIT_FAILURE);
        }
```

2. We've made sure the **fork** worked, so if **fork_result** equals zero, we're in the child process:

```
        if (fork_result == 0) {
            data_processed = read(file_pipes[0], buffer, BUFSIZ);
            printf("Read %d bytes: %s\n", data_processed, buffer);
            exit(EXIT_SUCCESS);
        }
```

3. Otherwise, we must be the parent process:

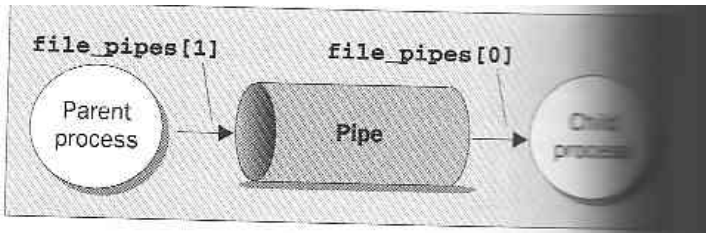
```
        else {
            data_processed = write(file_pipes[1], some_data,
                                   strlen(some_data));
            printf("Wrote %d bytes\n", data_processed);
        }
    }
    exit(EXIT_SUCCESS);
}
```

When we run this progra, the output is, as before:

```
$ pipe2
Wrote 3 bytes
Read 3 bytes: 123
```

How It Works

The program creates a pipe with the **pipe** call. It then uses the **fork** call to create a new process. The parent writes to the pipe and the child read from the pipe.



Parent and Child Processes

The child process can be a different program than the parent.

Try It Out - Pipes and exec

Here we have a data producer program and a data consumer program.

1. For the first program, we adapt **pipe2.c** to **pipe3.c**. The lines that we've changed are shown shaded:

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

int main()
{
    int data_processed;
    int file_pipes[2];
    const char some_data[] = "123";
    char buffer[BUFSIZ + 1];
    int fork_result;

    memset(buffer, '\0', sizeof(buffer));

    if (pipe(file_pipes) == 0) {
        fork_result = fork();
        if (fork_result == -1) {
            fprintf(stderr, "Fork failure");
            exit(EXIT_FAILURE);
        }

        if (fork_result == 0) {
            sprintf(buffer, "%d", file_pipes[0]);
            (void)execl("pipe4", "pipe4", buffer, (char *)0);
            exit(EXIT_FAILURE);
        }
        else {
            data_processed = write(file_pipes[1], some_data,
                                   strlen(some_data));
            printf("%d - wrote %d bytes\n", getpid(), data_processed);
        }
    }
    exit(EXIT_SUCCESS);
}
```

2. The 'consumer' program, **pipe4.c**, that reads the data is much similer:

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

int main(int argc, char *argv[])
{
    int data_processed;
    char buffer[BUFSIZ + 1];
    int file_descriptor;

    memset(buffer, '\0', sizeof(buffer));
    sscanf(argv[1], "%d", &file_descriptor);
    data_processed = read(file_descriptor, buffer, BUFSIZ);

    printf("%d - read %d bytes: %s\n", getpid(), data_processed, buffer);
    exit(EXIT_SUCCESS);
}
```

Remembering that **pipe3** invokes the **pipe4** program for us, when we run **pipe3**, we get the following output:

```
$ pipe3
980 - wrote 3 bytes
981 - read 3 bytes: 123
```

How It Works

The **pipe3** program uses the **pipe** call to create a pippe and then using the **fork** call to create a new process.

pipe4 receives the descriptor number of the pipe as an argument.

A call to **execl** is used to invoke the **pipe4** program. The arguments to **execl** are:

- The program to invoke.
- `argv[0]`, which takes the program name.
- `argv[1]`, which contains the file descriptor number we want the program to read from.
- `(char *)0`, which terminates the parameters.

Reading Closed Pipes

A **read** on a pipe that isn't open for writing will return 0, allowing the reading process to avoid the 'blocked forever' condition.

Pipes used as Standard Input and Output

We can arange for one of the pipe file descriptors to have a known value, usually the standard input, 0, or the standard output, 1.

The advantage is that we can invoke standard programs, ones that don't expect a file descriptor as a parameter.

There are two closedly related versions of **dup**, that have the prototypes:

```
#include <unistd.h>

int dup(int file_descriptor);
int dup2(int file_descriptor_one, int file_descriptor_two);
```

We can get the same effect as `dup` and `dup2` by using the more general `fcntl` call, with a command `F_DUPFD`. Having said that, the `dup` call is easier to use, since it's tailored specifically to the needs of creating duplicate file descriptors. It's also very commonly used, so you'll find it more frequently in existing programs than `fcntl` and `F_DUPFD`.

File Descriptor Manipulation by close and dup

The **dup** always returns a new file descriptor using the lowest available number.

By first closing file descriptor 0 and then calling **dup**, the new file descriptor will have the number zero.

File descriptor number	Initially	After close	After dup
0	Standard input		Pipe file descriptor
1	Standard output	Standard output	Standard output
2	Standard error	Standard error	Standard error
3	Pipe file descriptor	Pipe file descriptor	Pipe file descriptor

Try It Out - Pipes and dup

1. Modify **pipe3.c** to **pipe5.c**, using the following code:

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

int main()
{
    int data_processed;
    int file_pipes[2];
    const char some_data[] = "123";
    int fork_result;

    if (pipe(file_pipes) == 0) {
        fork_result = fork();
        if (fork_result == -1) {
            fprintf(stderr, "Fork failure");
            exit(EXIT_FAILURE);
        }

        if (fork_result == 0) {
            close(0);
            dup(file_pipes[0]);
            close(file_pipes[0]);
            close(file_pipes[1]);

            execlp("od", "od", "-c", (char *)0);
            exit(EXIT_FAILURE);
        }
        else {
            close(file_pipes[0]);
            data_processed = write(file_pipes[1], some_data,
                                   strlen(some_data));
            close(file_pipes[1]);
            printf("%d - wrote %d bytes\n", getpid(), data_processed);
        }
    }
    exit(EXIT_SUCCESS);
}
```

The output from this program is:

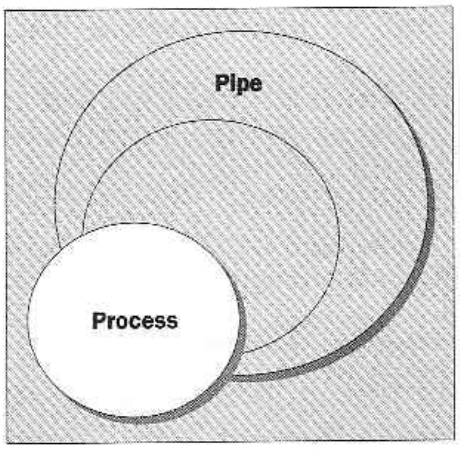
```
$ pipe5
1239 - wrote 3 bytes
0000000   1   2   3
0000003
```

How It Works

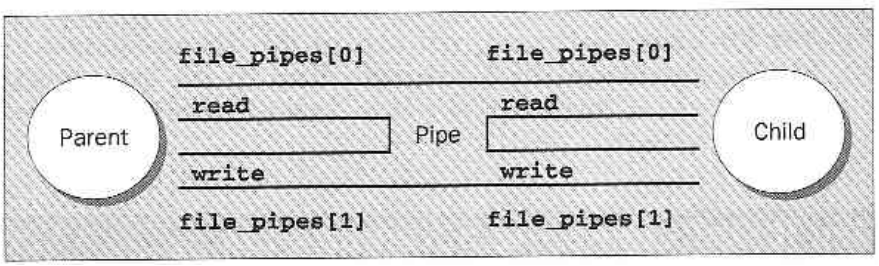
The program creates a pipe and then forks, creating a child process.

The parent and child have access to the pipe.

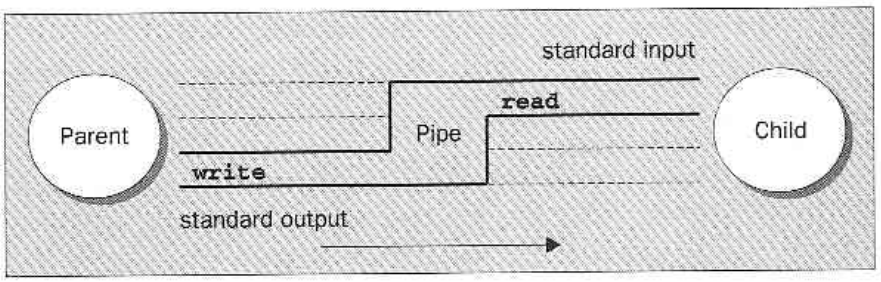
We can show the sequence pictorially. After the call to **pipe**:



After the call to **fork**:



When the program is rady to transfer data:



Named Pipes: FIFOs

We can exchange data with **FIFOs**, often referred to as **named pipes**.

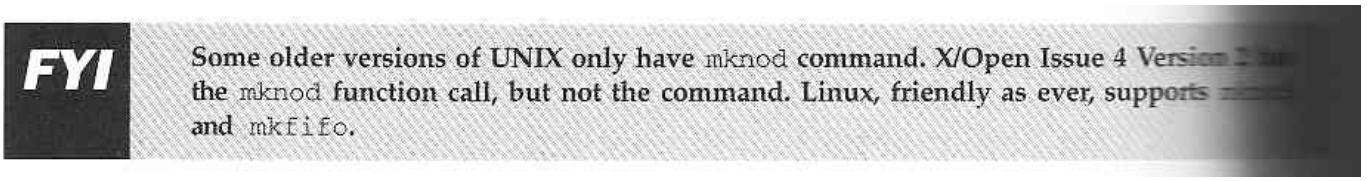
A named pipe is a special type of file that exists as a name in the file system, but behaves like the unnamed pipes that we've met already.

We can create a named pipe using the old UNIX **mknod** command:

```
$ mknod filename p
```

However, it is not in X/Open/ command list, so we use the **mkfifo** command:

```
$ mkfifo filename
```



From inside a program, we can use two different calls. These are:

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

int mkfifo(const char *filename, mode_t mode);
int mknod(const char *filename, mode_t mode | S_IFIFO, (dev_t) 0);
```

Try It Out - Creating a Named Pipe

For **fifo1.c**, just type in the following code:

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

int main()
{
    int res = mkfifo("/tmp/my_fifo", 0777);
    if (res == 0) printf("FIFO created\n");
    exit(EXIT_SUCCESS);
}
```

We can look for the pipe with:

```
$ ls -lF /tmp/my_fifo
prwxr-xr-x   1 rick     users           0 Dec 10 14:55 /tmp/my_fifo|
```

How It Works

The program uses the **mkfifo** function to create a special file.

Accessing a FIFO

One very useful feature of named pipes is that, because they appear in the file system, we can use them in commands where we would normally use a file name.

Try It Out - Accessing a FIFO File

1. First, let's try reading the (empty) FIFO:

```
$ cat < /tmp/my_fifo
```

2. Now try writing to the FIFO:

```
$ echo "sdsdfasdf" > /tmp/my_fifo
```

3. If we do both at once, we can pass information through the pipe:

```
$ cat < /tmp/my_fifo &
[1] 1316
$ echo "sdsdfasdf" > /tmp/my_fifo
sdsdfasdf

[1]+  Done                    cat </tmp/my_fifo
$
```

NOTICE: the first two stages simply hang until we interrupt them with Ctrl-C.

How It Works

Since there was no data in the FIFO, the **cat** and **echo** programs blocks, waiting for some data to arrive and some other process to read the data, respectively.

The thrid stage works as expected.

Unlike a pipe created with the **pipe** *call, a FIFO exists as a named file, not as an open file descriptor, and must be opened before it can be read from or written to. You open and close a FIFO using the same* **open** *and* **close** *functions that we saw used earlier for files, with some additional functionality. The* **open** *call is passed the path name of the FIFO, rather than that of a regular file.*

Opening a FIFO with open

The main restriction on opening FIFOs is that a program may not open a FIFO for reading and writing with the mode **O_RDWR**.

A process will read its own output back from a pipe if it were opened read/write.

There are four legal combinations of **O_RDONLY**, **O_WRONLY** and the **O_NONBLOCK** flag. We'll consider each in turn.

```
open(const char *path, O_RDONLY);
```

In this case, the **open** call will block, i.e. not return until a process opens the same FIFO for writing.

```
open(const char *path, O_RDONLY | O_NONBLOCK);
```

The **open** will now succeed and return immediately, even if the FIFO has not been opened for writing by any process.

```
open(const char *path, O_WRONLY);
```

In this case, the **open** call will block until a process opens the same FIFO for reading.

```
open(const char *path, O_WRONLY | O_NONBLOCK);
```

This wil always return immediately, but if no process has the FIFO open for reading, **open** will return an error, -1, and the FIFO won't be opened.

Try It Out - Opening FIFO Files

1. Start with the header files, a **#define** and the check that the correct number of command-line arguments have been supplied:

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

#define FIFO_NAME "/tmp/my_fifo"

int main(int argc, char *argv[])
{
    int res;
    int open_mode = 0;

    if (argc < 2) {
        fprintf(stderr, "Usage: %s <some combination of\
              O_RDONLY O_WRONLY O_NONBLOCK\n", *argv);
        exit(EXIT_FAILURE);
    }
```

```
    argv++;
    if (strncmp(*argv, "O_RDONLY", 8) == 0) open_mode |= O_RDONLY;
    if (strncmp(*argv, "O_WRONLY", 8) == 0) open_mode |= O_WRONLY;
    if (strncmp(*argv, "O_NONBLOCK", 10) == 0) open_mode |= O_NONBLOCK;
    argv++;
```

2. Assuming that the program passed the test, we now set the value of **open_mode** from those arguments:

```
    if (*argv) {
        if (strncmp(*argv, "O_RDONLY", 8) == 0) open_mode |= O_RDONLY;
        if (strncmp(*argv, "O_WRONLY", 8) == 0) open_mode |= O_WRONLY;
        if (strncmp(*argv, "O_NONBLOCK", 10) == 0) open_mode |= O_NONBLOCK;
    }
```

3. We now check whether the FIFO exists, create it if nmecessayr, open it and give it output, wait, and close it.

```
    if (access(FIFO_NAME, F_OK) == -1) {
        res = mkfifo(FIFO_NAME, 0777);
        if (res != 0) {
            fprintf(stderr, "Could not create fifo %s\n", FIFO_NAME);
            exit(EXIT_FAILURE);
        }
    }

    printf("Process %d opening FIFO\n", getpid());
    res = open(FIFO_NAME, open_mode);
    printf("Process %d result %d\n", getpid(), res);
    sleep(5);
    if (res != -1) (void)close(res);
    printf("Process %d finished\n", getpid());
    exit(EXIT_SUCCESS);
}
```

How It Works

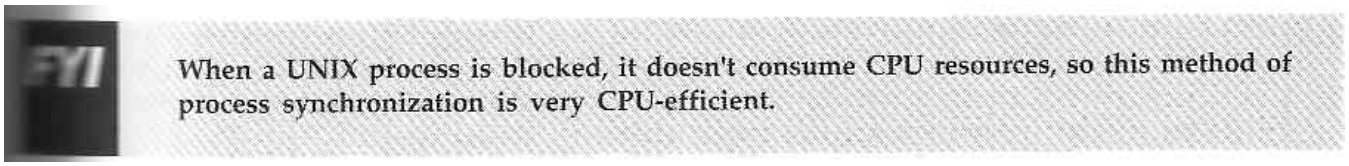
This program allows us to specify on the command line the combination of **O_RDONLY**, **O_WRONLY** and **O_NONBLOCK** that we wish to use.

O_RDONLY and O_WRONLY with no O_NONBLOCK

Let's try out a couple of combinations.

```
$ ./fifo2 O_RDONLY &
[1] 152
Process 152 opening FIFO
$ ./fifo2 O_WRONLY
Process 153 opening FIFO
Process 152 result 3
Process 153 result 3
Process 152 finished
Process 153 finished
```

It allows the reader process to start, wait in the **open** command and then both programs to continue when the second program opens the FIFO.



Here is another combination:

```
$ ./fifo2 O_RDONLY O_NONBLOCK &
[1] 160
Process 160 opening FIFO
Process 160 result 3
$ ./fifo2 O_WRONLY
Process 161 opening FIFO
Process 161 result 3
Process 160 finished
Process 161 finished
[1]+  Done                    fifo2 O_RDONLY O_NONBLOCK
```

This time, the reader process executes the **open** call and continues immediately, even though no writer process is present.

Reading and Writing FIFOs

Using the **O_NONBLOCK** mode affects how **read** and **write** calls behave on FIFOs.

A **read** on an empty blocking FIFO will wait until some data can be read.

A **write** on a full blocking FIFO will wait until the data can be written.

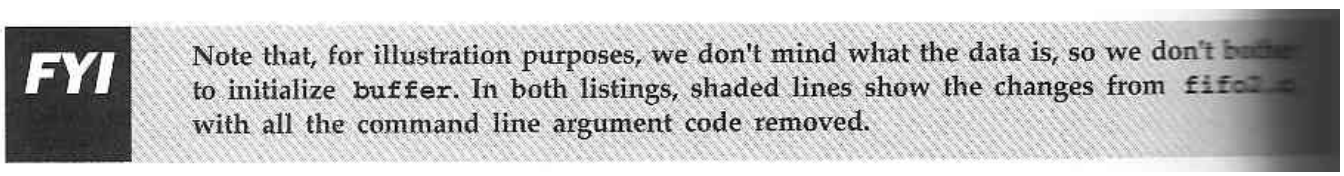
A **write** on a FIFO that can't accept all of the bytes being written will either:

- Fail if the request is for **PIPE_BUF** bytes or less and the data can't be written.
- Write part of the data if the request is for more than **PIPE_BUF** bytes, returning the number of bytes actually written, which could be zero.

Try It Out - Inter-process Communication with FIFOs

To show how unrelated processes can communicate using named pipes, we need two separate program, **fifo3.c** and **fifo4.c**.

1. The first program is our producer program. This creates the pipe if required, then writes data to it as quickly as possilbe.



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

#define FIFO_NAME "/tmp/my_fifo"
#define BUFFER_SIZE PIPE_BUF
#define TEN_MEG (1024 * 1024 * 10)

int main()
{
    int pipe_fd;
    int res;
    int open_mode = O_WRONLY;
    int bytes_sent = 0;
    char buffer[BUFFER_SIZE + 1];

    if (access(FIFO_NAME, F_OK) == -1) {
        res = mkfifo(FIFO_NAME, 0777);
        if (res != 0) {
            fprintf(stderr, "Could not create fifo %s\n", FIFO_NAME);
            exit(EXIT_FAILURE);
        }
    }

    printf("Process %d opening FIFO O_WRONLY\n", getpid());
    pipe_fd = open(FIFO_NAME, open_mode);
    printf("Process %d result %d\n", getpid(), pipe_fd);

    if (pipe_fd != -1) {
        while(bytes_sent < TEN_MEG) {
            res = write(pipe_fd, buffer, BUFFER_SIZE);
            if (res == -1) {
                fprintf(stderr, "Write error on pipe\n");
                exit(EXIT_FAILURE);
            }
            bytes_sent += res;
        }
        (void)close(pipe_fd);
    }
    else {
        exit(EXIT_FAILURE);
    }

    printf("Process %d finished\n", getpid());
    exit(EXIT_SUCCESS);
}
```

2. Our second program, the consumer, is much simpler. It reads and discards data from the FIFO.

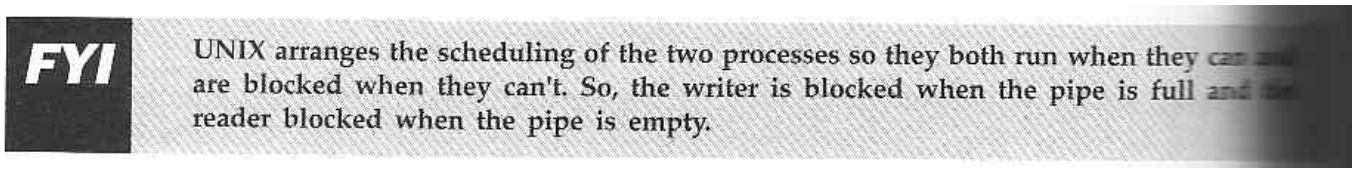
```
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <fcntl.h>
#include <limits.h>
#include <sys/types.h>
#include <sys/stat.h>

#define FIFO_NAME "/tmp/my_fifo"
#define BUFFER_SIZE PIPE_BUF

int main()
{
    int pipe_fd;
    int res;
    int open_mode = O_RDONLY;
    char buffer[BUFFER_SIZE + 1];
    int bytes_read = 0;

    memset(buffer, '\0', sizeof(buffer));

    printf("Process %d opening FIFO O_RDONLY\n", getpid())
    pipe_fd = open(FIFO_NAME, open_mode);
    printf("Process %d result %d\n", getpid(), pipe_fd);

    if (pipe_fd != -1) {
        do {
            res = read(pipe_fd, buffer, BUFFER_SIZE);
            bytes_read += res;
        } while (res > 0);
        (void)close(pipe_fd);
    }
    else {
        exit(EXIT_FAILURE);
    }

    printf("Process %d finished, %d bytes read\n", getpid(), bytes_read);
    exit(EXIT_SUCCESS);
}
```

When we run these porgrams at the same time, using the **time** command to time the reader, the output we get is:

```
$ ./fifo3 &
[1] 375
Process 375 opening FIFO O_WRONLY
$ time ./fifo4
Process 377 opening FIFO O_RDONLY
Process 375 result 3
Process 377 result 3
Process 375 finished
Process 377 finished, 10485760 bytes read
0.00user 0.42system 0:00.75elapsed 55%CPU (0avgtext+0avgdata 0maxresident
0inputs+0outputs (14major+10minor)pagefaults 0swaps
[1]+  Done                    fifo3
```

How It Works

Both programs use the FIFO in blocking mode. **fifo3** is started first and wait for the FIFO to open. When **fifo4** is started, the pipe is unblocked and data transfer occurs.



Advanced Topic: Client/Server using FIFOs

Now let's build a client/server application using named pipes.

We want to allow multiple client processes to send data to the server, using a single pipe.

Returning data to a client requires one pipe per client.

Try It Out - An Example Client/Server Application

1. First, we need a header file, **cliserv.h**, that defines the data common to both client and server programs.

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

#define SERVER_FIFO_NAME "/tmp/serv_fifo"
#define CLIENT_FIFO_NAME "/tmp/cli_%d_fifo"

#define BUFFER_SIZE 20

struct data_to_pass_st {
    pid_t  client_pid;
    char   some_data[BUFFER_SIZE - 1];
};
```

2. The server program, **server.c**, creates and opens a pipe

```
#include "cliserv.h"
#include <ctype.h>

int main()
{
    int server_fifo_fd, client_fifo_fd;
    struct data_to_pass_st my_data;
    int read_res;
    char client_fifo[256];
    char *tmp_char_ptr;

    mkfifo(SERVER_FIFO_NAME, 0777);
    server_fifo_fd = open(SERVER_FIFO_NAME, O_RDONLY);
    if (server_fifo_fd == -1) {
        fprintf(stderr, "Server fifo failure\n");
        exit(EXIT_FAILURE);
    }
```

```
    sleep(10); /* lets clients queue for demo purposes */

    do {
        read_res = read(server_fifo_fd, &my_data, sizeof(my_data));
        if (read_res > 0) {
```

3. We perform some processing on the data just read form the client.

```
            tmp_char_ptr = my_data.some_data;
            while (*tmp_char_ptr) {
                *tmp_char_ptr = toupper(*tmp_char_ptr);
                tmp_char_ptr++;
            }
            sprintf(client_fifo, CLIENT_FIFO_NAME, my_data.client_pid);
```

4. Then we send the processed back back, opeing the client pipe.

```
            client_fifo_fd = open(client_fifo, O_WRONLY);
            if (client_fifo_fd != -1) {
                write(client_fifo_fd, &my_data, sizeof(my_data));
                close(client_fifo_fd);
            }
        }
    } while (read_res > 0);
    close(server_fifo_fd);
    unlink(SERVER_FIFO_NAME);
    exit(EXIT_SUCCESS);
}
```

5. Here's the client, **client.c**. It opens the server FIFO and creates a client FIFO.

```
#include "cliserv.h"
#include <ctype.h>

int main()
{
    int server_fifo_fd, client_fifo_fd;
    struct data_to_pass_st my_data;
    int times_to_send;
    char client_fifo[256];

    server_fifo_fd = open(SERVER_FIFO_NAME, O_WRONLY);
    if (server_fifo_fd == -1) {
        fprintf(stderr, "Sorry, no server\n");
        exit(EXIT_FAILURE);
    }

    my_data.client_pid = getpid();
    sprintf(client_fifo, CLIENT_FIFO_NAME, my_data.client_pid);
    if (mkfifo(client_fifo, 0777) == -1) {
        fprintf(stderr, "Sorry, can't make %s\n", client_fifo);
        exit(EXIT_FAILURE);
    }
```

6. For each of the five loops, the client data is sent to the server. Finally the server FIFO is closed and the client FIFO is removed from memory.

```
    for (times_to_send = 0; times_to_send < 5; times_to_send++) {
        sprintf(my_data.some_data, "Hello from %d", my_data.client_pid);
        printf("%d sent %s, ", my_data.client_pid, my_data.some_data);
        write(server_fifo_fd, &my_data, sizeof(my_data));
        client_fifo_fd = open(client_fifo, O_RDONLY);
        if (client_fifo_fd != -1) {
            if (read(client_fifo_fd, &my_data, sizeof(my_data)) > 0) {
                printf("received: %s\n", my_data.some_data);
            }
            close(client_fifo_fd);
        }
    }
    close(server_fifo_fd);
    unlink(client_fifo);
    exit(EXIT_SUCCESS);
}
```

The following shell commands run a single copy of the server and several clients, to test out the programs.

```
$ server &
$ for i in 1 2 3 4 5
do
client &
done
```

This starts one server process and five client processes. Here is the output.

```
531 sent Hello from 531, received: HELLO FROM 531
532 sent Hello from 532, received: HELLO FROM 532
529 sent Hello from 529, received: HELLO FROM 529
530 sent Hello from 530, received: HELLO FROM 530
531 sent Hello from 531, received: HELLO FROM 531
532 sent Hello from 532, received: HELLO FROM 532
```

How It Works

The server creates its FIFO in read-only mode and blocks.

The client opens the FIFO for writing and createsits own uniquely-named FIFO for reading back from the server.

For a real server process that needed to wait for further clients, we ould need to modify it to either:

- Open a file descriptor to its own server pipe, so **read** always blocks rather than returning 0.
- When **read** returns 0 bytes, close and reopen the server pipe, so the server process blocks in the **open** waiting for a client, just as it did when it first started.

The CD Application

Now we make the CD application a simple client/server system. It takes serveral programs.

Here is the **Makefile** to show how the programs will fit together.

```
all:    server_app client_app

CC=gcc
CFLAGS=-pedantic -Wall

# For debugging un-comment the next line
# DFLAGS=-DDEBUG_TRACE=1 -g

.c.o:
     $(CC) $(CFLAGS) $(DFLAGS) -c $<
```

```
app_ui.o: app_ui.c cd_data.h
cd_dbm.o: cd_dbm.c cd_data.h
client_if.o: client_if.c cd_data.h cliserv.h
pipe_imp.o: pipe_imp.c cd_data.h cliserv.h
server.o: server.c cd_data.h cliserv.h

client_app: app_ui.o client_if.o pipe_imp.o
     $(CC) -o client_app  $(DFLAGS) app_ui.o client_if.o pipe_imp.o

server_app: server.o cd_dbm.o pipe_imp.o
     $(CC) -o server_app  $(DFLAGS) server.o cd_dbm.o pipe_imp.o -ldbm
```

Aims

Our aim is to split the part of the application that deals with the database away from the user interface part of the specification.

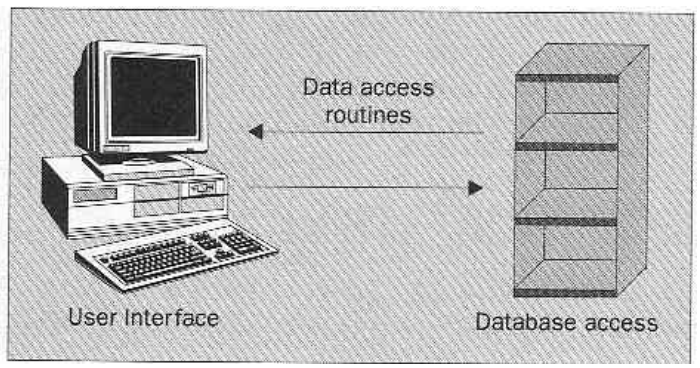
Implementation

In the earlier, single process version of the application we used a set of data access routines for manipulating the data. There were:

```
int database_initialize(const int new_database);
void database_close(void);
cdc_entry get_cdc_entry(const char *cd_catalog_ptr);
cdt_entry get_cdt_entry(const char *cd_catalog_ptr, const int track_no);
int add_cdc_entry(const cdc_entry entry_to_add);
int add_cdt_entry(const cdt_entry entry_to_add);
int del_cdc_entry(const char *cd_catalog_ptr);
int del_cdt_entry(const char *cd_catalog_ptr, const int track_no);
cdc_entry search_cdc_entry(const char *cd_catalog_ptr,
                           int *first_call_ptr);
```

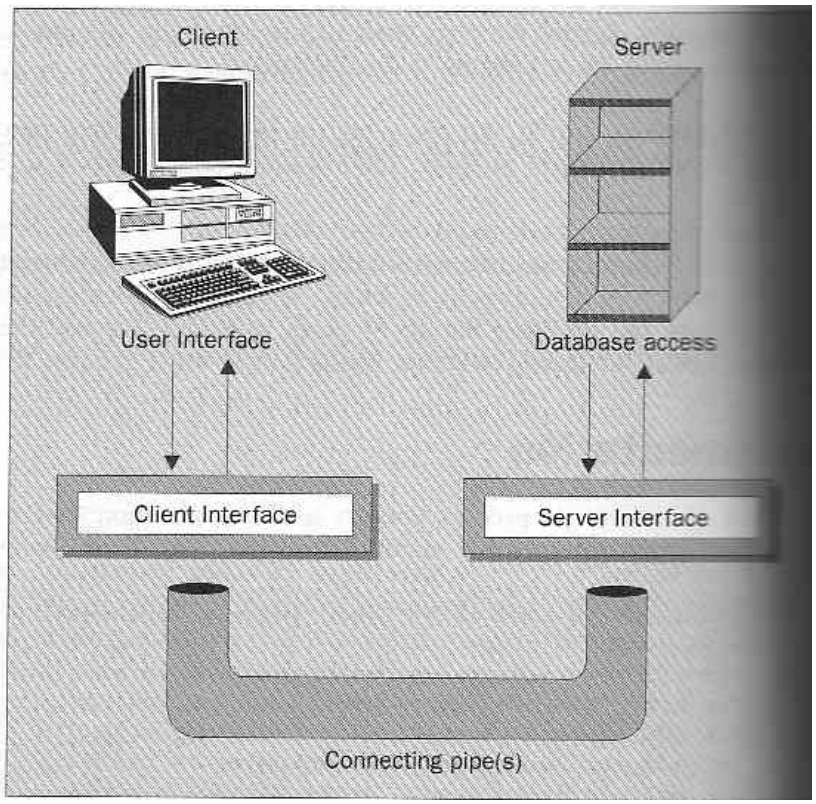
These functions provide a convient place to make a clean separation between client and server.

In the single process implementation, we can view the application as having two parts:

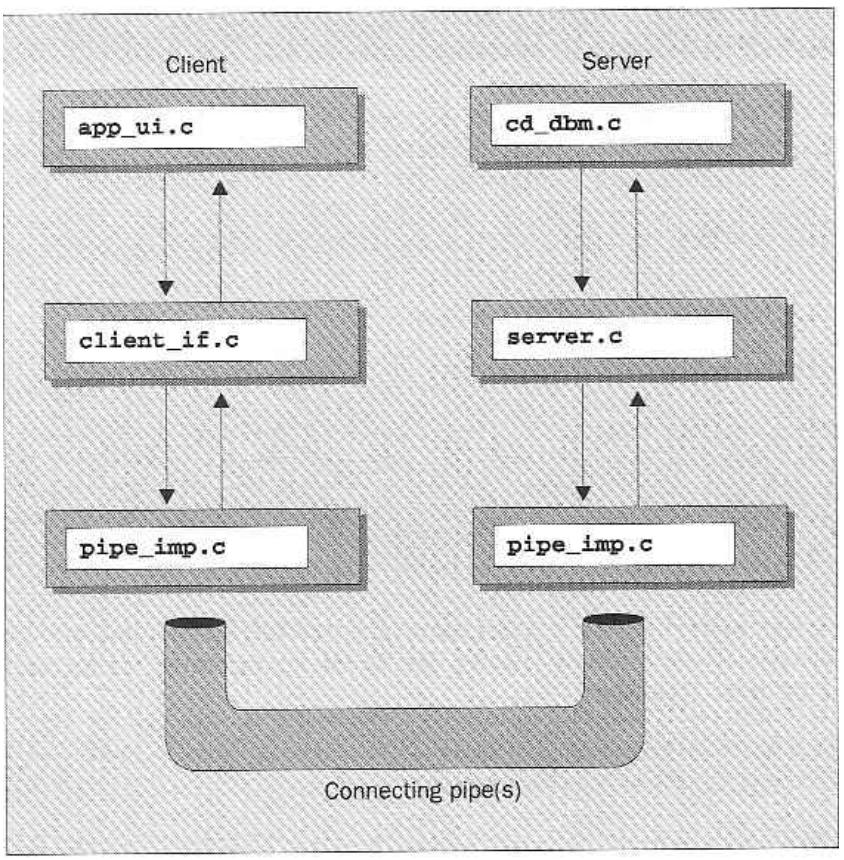


In the client server implementation, we want to logically insert some named pipes and supporting code between the two major parts of the application.

This is the structure we need:



In our implementation, we choose to put both the client and server interface routines in the same file.



We have six .c files

Some parts of this file are dependent on the specific client/server implementation, in this case named pipes. We'll be changing to a different client/server model at the end of the next chapter.

First, we'll look at **cliserv.h**. This file defines the client/server intrfaces. It is required by both client and server implementtions.

Try It Out - The header File, cliserv.h

1. First, we have a **#define** that includes the feature test marco **_POSIX_SOURCE**.

```
#define _POSIX_SOURCE
```

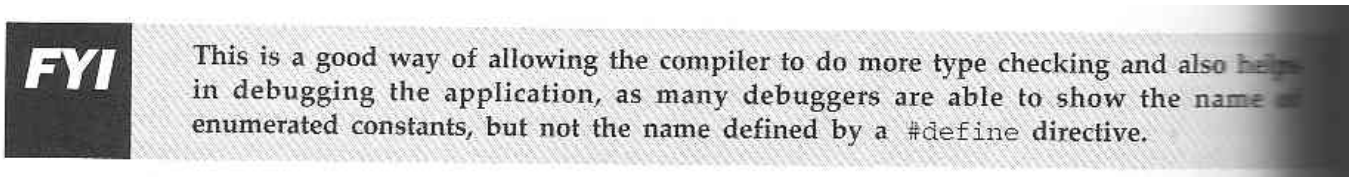
2. There follows the required **#include** headers:

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

3. We define the named pipes:

```
#define SERVER_PIPE "/tmp/server_pipe"
#define CLIENT_PIPE "/tmp/client_%d_pipe"

#define ERR_TEXT_LEN 80
```

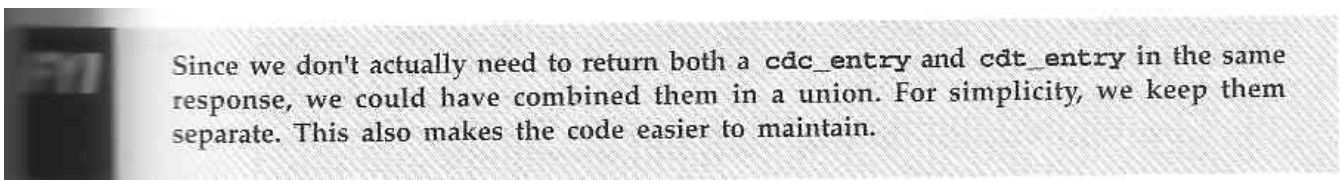
4. We implement the commands as enumberated types, rather then **#define**s.



The first **typedef** gives the type of request being sent to the server, the second the server response to the client.

```
typedef enum {
    s_create_new_database = 0,
    s_get_cdc_entry,
    s_get_cdt_entry,
    s_add_cdc_entry,
    s_add_cdt_entry,
    s_del_cdc_entry,
    s_del_cdt_entry,
    s_find_cdc_entry
} client_request_e;

typedef enum {
    r_success = 0,
    r_failure,
    r_find_no_more
} server_response_e;
```

5. Next, we declare a structure that will form the message passed in both directions between the two processes.



```
typedef struct {
    pid_t               client_pid;
    client_request_e    request;
    server_response_e   response;
    cdc_entry           cdc_entry_data;
    cdt_entry           cdt_entry_data;
    char                error_text[ERR_TEXT_LEN + 1];
} message_db_t;
```

6. Finally, we get the pipe interface functions that perofrm dta transfer.

```
int server_starting(void);
void server_ending(void);
int read_request_from_client(message_db_t *rec_ptr);
int start_resp_to_client(const message_db_t mess_to_send);
int send_resp_to_client(const message_db_t mess_to_send);
void end_resp_to_client(void);

int client_starting(void);
void client_ending(void);
int send_mess_to_server(message_db_t mess_to_send);
int start_resp_from_server(void);
int read_resp_from_server(message_db_t *rec_ptr);
void end_resp_from_server(void);
```

Client Interface Function

Now we look at **client_if.c**. This provides 'fake' versions fo the database access routines.

Try It Out -The Client's Interpreter

1. This file implements the nine database functions protyped in **cd_data.h**.

The file starts with **#include** files and constants:

```
#define _POSIX_SOURCE

#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>
#include <fcntl.h>
#include <limits.h>
#include <sys/types.h>
#include <sys/stat.h>

#include "cd_data.h"
#include "cliserv.h"
```

2. The static variable **mypid/B> reduces the number of calls to getpid that would otherwise be require.**

```
static pid_t mypid;

static int read_one_response(message_db_t *rec_ptr);
```

3. The database_initialize and _close routines are still called but now use pipes.

```
int database_initialize(const int new_database)
{
    if (!client_starting()) return(0);
    mypid = getpid();
    return(1);
}

void database_close(void) {
    client_ending();
}
```

4. The **get_cdc_entry** routine is called to get a catalog entry from the database, given a CD catalog title.

```
cdc_entry get_cdc_entry(const char *cd_catalog_ptr)
{
    cdc_entry ret_val;
    message_db_t mess_send;
    message_db_t mess_ret;
```

```
    ret_val.catalog[0] = '\0';
    mess_send.client_pid = mypid;
    mess_send.request = s_get_cdc_entry;
    strcpy(mess_send.cdc_entry_data.catalog, cd_catalog_ptr);

    if (send_mess_to_server(mess_send)) {
        if (read_one_response(&mess_ret)) {
            if (mess_ret.response == r_success) {
                ret_val = mess_ret.cdc_entry_data;
            } else {
                fprintf(stderr, mess_ret.error_text);
            }
        } else {
            fprintf(stderr, "Server failed to respond\n");
        }
    } else {
        fprintf(stderr, "Server not accepting requests\n");
    }
    return(ret_val);
}
```

5. Here's the source for the function **read_one_response** that we use to avoid duplicationg code:

```
static int read_one_response(message_db_t *rec_ptr)
{
    int return_code = 0;
    if (!rec_ptr) return(0);

    if (start_resp_from_server()) {
        if (read_resp_from_server(rec_ptr)) {
            return_code = 1;
        }
        end_resp_from_server();
    }
    return(return_code);
}
```

6. The other **get_xxx, del_xx**, and **add_xxx** routines are implemented in a similar way to the **get_cdc_entry**.

```
cdt_entry get_cdt_entry(const char *cd_catalog_ptr, const int track_no)
{
    cdt_entry ret_val;
    message_db_t mess_send;
    message_db_t mess_ret;

    ret_val.catalog[0] = '\0';
    mess_send.client_pid = mypid;
    mess_send.request = s_get_cdt_entry;
    strcpy(mess_send.cdt_entry_data.catalog, cd_catalog_ptr);
    mess_send.cdt_entry_data.track_no = track_no;

    if (send_mess_to_server(mess_send)) {
        if (read_one_response(&mess_ret)) {
            if (mess_ret.response == r_success) {
                ret_val = mess_ret.cdt_entry_data;
            } else {
                fprintf(stderr, mess_ret.error_text);
            }
        } else {
            fprintf(stderr, "Server failed to respond\n");
        }
    } else {
        fprintf(stderr, "Server not accepting requests\n");
    }
    return(ret_val);
}
```

7. Next, two functions for adding data, first to the catalog and then to the tracks database:

```
int add_cdc_entry(const cdc_entry entry_to_add)
{
    message_db_t mess_send;
    message_db_t mess_ret;

    mess_send.client_pid = mypid;
    mess_send.request = s_add_cdc_entry;
    mess_send.cdc_entry_data = entry_to_add;

    if (send_mess_to_server(mess_send)) {
        if (read_one_response(&mess_ret)) {
            if (mess_ret.response == r_success) {
                return(1);
            } else {
                fprintf(stderr, mess_ret.error_text);
            }
        } else {
            fprintf(stderr, "Server failed to respond\n");
        }
    } else {
        fprintf(stderr, "Server not accepting requests\n");
    }
    return(0);
}
```

```
int add_cdt_entry(const cdt_entry entry_to_add)
{
    message_db_t mess_send;
    message_db_t mess_ret;

    mess_send.client_pid = mypid;
    mess_send.request = s_add_cdt_entry;
    mess_send.cdt_entry_data = entry_to_add;

    if (send_mess_to_server(mess_send)) {
        if (read_one_response(&mess_ret)) {
            if (mess_ret.response == r_success) {
                return(1);
            } else {
                fprintf(stderr, mess_ret.error_text);
            }
        } else {
            fprintf(stderr, "Server failed to respond\n");
        }
    } else {
        fprintf(stderr, "Server not accepting requests\n");
    }
    return(0);
}
```

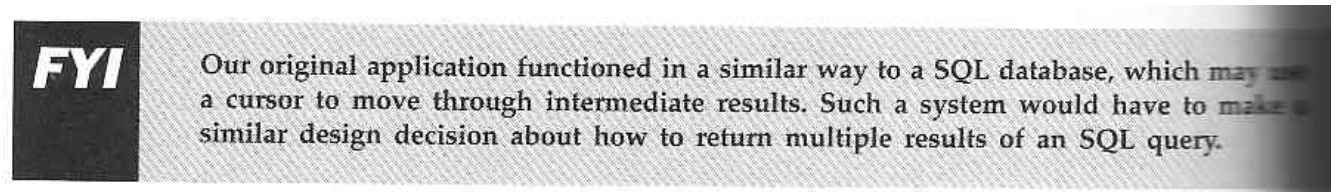
8. And lastly, two functions for data deletion:

```
int del_cdc_entry(const char *cd_catalog_ptr)
{
    message_db_t mess_send;
    message_db_t mess_ret;

    mess_send.client_pid = mypid;
    mess_send.request = s_del_cdc_entry;
    strcpy(mess_send.cdc_entry_data.catalog, cd_catalog_ptr);

    if (send_mess_to_server(mess_send)) {
        if (read_one_response(&mess_ret)) {
            if (mess_ret.response == r_success) {
                return(1);
            } else {
                fprintf(stderr, mess_ret.error_text);
            }
        } else {
            fprintf(stderr, "Server failed to respond\n");
        }
    } else {
        fprintf(stderr, "Server not accepting requests\n");
    }
    return(0);
}
```

```
int del_cdt_entry(const char *cd_catalog_ptr, const int track_no)
{
    message_db_t mess_send;
    message_db_t mess_ret;

    mess_send.client_pid = mypid;
    mess_send.request = s_del_cdt_entry;
     strcpy(mess_send.cdt_entry_data.catalog, cd_catalog_ptr);
    mess_send.cdt_entry_data.track_no = track_no;

    if (send_mess_to_server(mess_send)) {
        if (read_one_response(&mess_ret)) {
            if (mess_ret.response == r_success) {
                return(1);
            } else {
                fprintf(stderr, mess_ret.error_text);
            }
        } else {
            fprintf(stderr, "Server failed to respond\n");
        }
    } else {
            fprintf(stderr, "Server not accepting requests\n");
    }
    return(0);
}
```

Searching the Database

The function for search on the CD key is rather more complex. We arrange for the server to return all the possible matches to a search and then store them in a temporary file until they are requested by the client.



Try It Out - Searchng

1. This calls three pipe functions that are used in the next section, **send_mess_to_server, start__resp_from_serever, and read_resp_from_server**:

```
cdc_entry search_cdc_entry(const char *cd_catalog_ptr, int *first_call_ptr)
{
    message_db_t mess_send;
    message_db_t mess_ret;

    static FILE *work_file = (FILE *)0;
    static int entries_matching = 0;
    cdc_entry ret_val;

    ret_val.catalog[0] = '\0';

    if (!work_file && (*first_call_ptr == 0)) return(ret_val);
```

2. Here's the first call to search, i.e. with ***first__call_ptr** set to **true**.

```
if (*first_call_ptr) {
    *first_call_ptr = 0;
    if (work_file) fclose(work_file);
    work_file = tmpfile();
    if (!work_file) return(ret_val);

    mess_send.client_pid = mypid;
    mess_send.request = s_find_cdc_entry;
    strcpy(mess_send.cdc_entry_data.catalog, cd_catalog_ptr);
```

3. Now there's this three-deep condition test, which makes calls to functions in **pipe_imp.c**.

```
    if (send_mess_to_server(mess_send)) {
        if (start_resp_from_server()) {
            while (read_resp_from_server(&mess_ret)) {
                if (mess_ret.response == r_success) {
                    fwrite(&mess_ret.cdc_entry_data,
                            sizeof(cdc_entry), 1, work_file);
                    entries_matching++;
                } else {
                    break;
                }
            } /* while */
        } else {
            fprintf(stderr, "Server not responding\n");
        }
    } else {
        fprintf(stderr, "Server not accepting requests\n");
    }
```

4. The next test checks whether the search had any luck. Then the **fseek** call sets the **work_file** to the next place for data to be written.

```
    if (entries_matching == 0) {
        fclose(work_file);
        work_file = (FILE *)0;
        return(ret_val);
    }
    (void)fseek(work_file, 0L, SEEK_SET);
```

5. This checks whether there are any matches left.

```
} else {
            if (entries_matching == 0) {
        fclose(work_file);
        work_file = (FILE *)0;
        return(ret_val);
    }
}
```

```
fread(&ret_val, sizeof(cdc_entry), 1, work_file);
entries_matching--;

return(ret_val);
```

The Server Interface

The server side needs a program to control the (renamed) **cd_access.c**, now **cd_dbm.c**.

The server's **main** function os listed here

Try It Out - server.c

1. First, the usual headers and the macro **_POSIX_SOURCE**.

```
#define _POSIX_SOURCE

#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>
#include <fcntl.h>
#include <limits.h>
#include <signal.h>
#include <string.h>
#include <sys/types.h>
#include <sys/stat.h>

#include "cd_data.h"
#include "cliserv.h"

int save_errno;
static int server_running = 1;

static void process_command(const message_db_t mess_command);

void catch_signals()
{
    server_running = 0;
}
```

2. Now comes the **main** function.

```
int main(int argc, char *argv[]) {
     struct sigaction new_action, old_action;
     message_db_t mess_command;
     int database_init_type = 0;
```

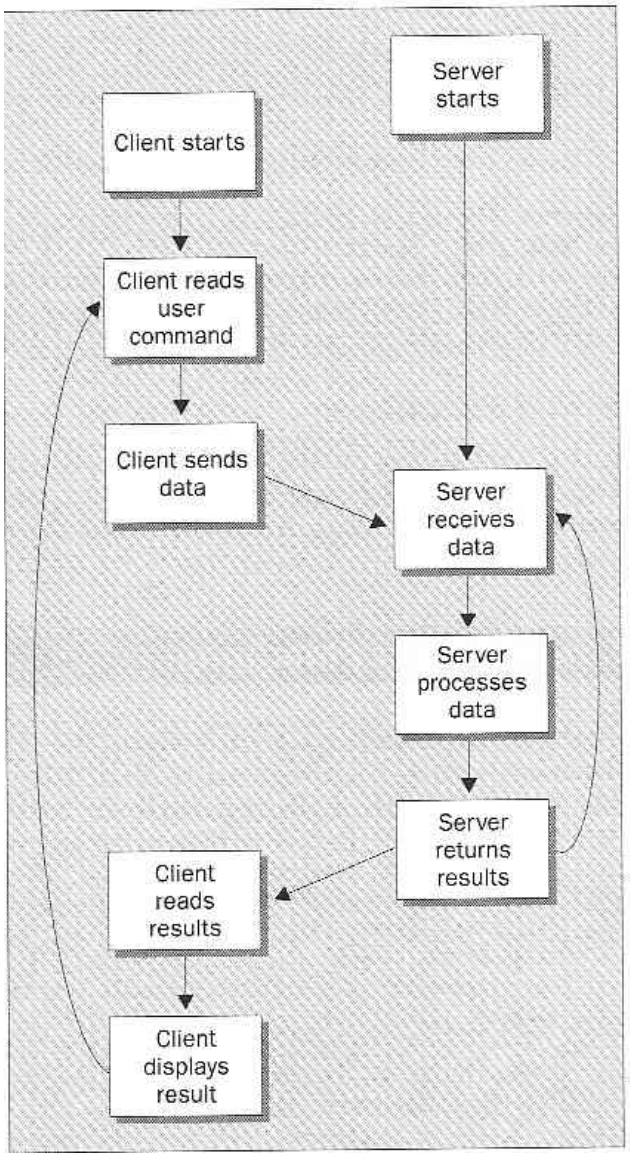
```
    new_action.sa_handler = catch_signals;
    sigemptyset(&new_action.sa_mask);
    new_action.sa_flags = 0;
    if ((sigaction(SIGINT, &new_action, &old_action) != 0) ||
        (sigaction(SIGHUP, &new_action, &old_action) != 0) ||
        (sigaction(SIGTERM, &new_action, &old_action) != 0)) {
        fprintf(stderr, "Server startup error, signal catching failed\n");
        exit(EXIT_FAILURE);
    }

    if (argc > 1) {
        argv++;
        if (strncmp("-i", *argv, 2) == 0) database_init_type = 1;
    }
    if (!database_initialize(database_init_type)) {
                fprintf(stderr, "Server error:-\
                        could not initialize database\n");
                exit(EXIT_FAILURE);
    }

    if (!server_starting()) exit(EXIT_FAILURE);

    while(server_running) {
        if (read_request_from_client(&mess_command)) {
            process_command(mess_command);
        } else {
            if(server_running) fprintf(stderr, "Server ended - can not \
                                        read pipe\n");
            server_running = 0;
        }
    } /* while */
    server_ending();
    exit(EXIT_SUCCESS);
}
```

3. Any client messages are fed to the **process_command** function.

```
static void process_command(const message_db_t comm)
{
    message_db_t resp;
    int first_time = 1;

    resp = comm; /* copy command back, then change resp as required */

    if (!start_resp_to_client(resp)) {
        fprintf(stderr, "Server Warning:-\
                 start_resp_to_client %d failed\n", resp.client_pid);
        return;
    }

    resp.response = r_success;
    memset(resp.error_text, '\0', sizeof(resp.error_text));
    save_errno = 0;

    switch(resp.request) {
        case s_create_new_database:
```

```
            if (!database_initialize(1)) resp.response = r_failure;
            break;
        case s_get_cdc_entry:
            resp.cdc_entry_data =
                        get_cdc_entry(comm.cdc_entry_data.catalog);
            break;
        case s_get_cdt_entry:
            resp.cdt_entry_data =
                        get_cdt_entry(comm.cdt_entry_data.catalog,
                                      comm.cdt_entry_data.track_no);
            break;
        case s_add_cdc_entry:
            if (!add_cdc_entry(comm.cdc_entry_data)) resp.response =
                        r_failure;
            break;
        case s_add_cdt_entry:
            if (!add_cdt_entry(comm.cdt_entry_data)) resp.response =
                        r_failure;
            break;
        case s_del_cdc_entry:
            if (!del_cdc_entry(comm.cdc_entry_data.catalog)) resp.response
                    = r_failure;
            break;
        case s_del_cdt_entry:
            if (!del_cdt_entry(comm.cdt_entry_data.catalog,
                comm.cdt_entry_data.track_no)) resp.response = r_failure;
            break;
        case s_find_cdc_entry:
            do {
                resp.cdc_entry_data =
                        search_cdc_entry(comm.cdc_entry_data.catalog,
                                         &first_time);
                if (resp.cdc_entry_data.catalog[0] != 0) {
                    resp.response = r_success;
                    if (!send_resp_to_client(resp)) {
                        fprintf(stderr, "Server Warning:-\
                            failed to respond to %d\n", resp.client_pid);
                        break;
                    }
                } else {
                    resp.response = r_find_no_more;
                }
            } while (resp.response == r_success);
        break;
        default:
            resp.response = r_failure;
            break;
    } /* switch */

    sprintf(resp.error_text, "Command failed:\n\t%s\n",
            strerror(save_errno));

    if (!send_resp_to_client(resp)) {
        fprintf(stderr, "Server Warning:-\
                failed to respond to %d\n", resp.client_pid);
    }

    end_resp_to_client();

    return;
}
```

Here is a diagram of the sequence of events that occur to pass data between client and server.



The Pipe

Here's the pipe implementation file, **pipe_imp.c**, whic has both the client- and server-side functions.

As we saw in Chapter 9, the symbol, **DEBUG_TRACE**, *can be defined to show the sequence of calls as the client and server processes pass messages between each other.*

Try It Out - Pipes Implementation Header

1. First the **#include**s:

```
#include "cd_data.h"
#include "cliserv.h"
```

2. We also define some values that we need in different functions within the file:.

```
static int server_fd = -1;
static pid_t mypid = 0;
static char client_pipe_name[PATH_MAX + 1] = {'\0'};
static int client_fd = -1;
static int client_write_fd = -1;
```

Server-side Function

Now we look at the server-side functions.

Try It Out - Server Functions

1. The **server_starting** routine creates the named pipe form which the server will read commands.

```
int server_starting(void)
{
    #if DEBUG_TRACE
        printf("%d :- server_starting()\n",  getpid());
    #endif

    (void)unlink(SERVER_PIPE);
    if (mkfifo(SERVER_PIPE, 0777) == -1) {
        fprintf(stderr, "Server startup error, no FIFO created\n");
        return(0);
    }

    if ((server_fd = open(SERVER_PIPE, O_RDONLY)) == -1) {
        if (errno == EINTR) return(0);
        fprintf(stderr, "Server startup error, no FIFO opened\n");
        return(0);
    }
    return(1);
}
```

2. When the server ends, it removes the named pipe, so clients can detect that no server is running

```
void server_ending(void)
{
    #if DEBUG_TRACE
        printf("%d :- server_ending()\n",  getpid());
    #endif

    (void)close(server_fd);
    (void)unlink(SERVER_PIPE);
}
```

3. The **read_request_from_client** function, shown below, will block reading the server pipe until a client writes a message into it:

```
int read_request_from_client(message_db_t *rec_ptr)
{
    int return_code = 0;
    int read_bytes;

    #if DEBUG_TRACE
        printf("%d :- read_request_from_client()\n",  getpid());
    #endif

    if (server_fd != -1) {
        read_bytes = read(server_fd, rec_ptr, sizeof(*rec_ptr));
```

...

```
    }
    return(return_code);
}
```

4. In the special case when no clients have the pipe open for writing, the **read** will return 0, i.e. it detects an **EOF**.

```
        if (read_bytes == 0) {
            (void)close(server_fd);
            if ((server_fd = open(SERVER_PIPE, O_RDONLY)) == -1) {
                if (errno != EINTR) {
                    fprintf(stderr, "Server error, FIFO open failed\n");
                }
                return(0);
            }
            read_bytes = read(server_fd, rec_ptr, sizeof(*rec_ptr));
        }
        if (read_bytes == sizeof(*rec_ptr)) return_code = 1;
```

Try It Out - Plumbing the Pipes

1. First, we open the client pipe:

```
int start_resp_to_client(const message_db_t mess_to_send)
{
    #if DEBUG_TRACE
        printf("%d :- start_resp_to_client()\n", getpid());
    #endif

    (void)sprintf(client_pipe_name, CLIENT_PIPE, mess_to_send.client_pid);
    if ((client_fd = open(client_pipe_name, O_WRONLY)) == -1) return(0);
    return(1);
}
```

2. The messages are all sent using calls to this function.

```
int send_resp_to_client(const message_db_t mess_to_send)
{
    int write_bytes;

    #if DEBUG_TRACE
        printf("%d :- send_resp_to_client()\n", getpid());
    #endif

    if (client_fd == -1) return(0);
    write_bytes = write(client_fd, &mess_to_send, sizeof(mess_to_send));
    if (write_bytes != sizeof(mess_to_send)) return(0);
    return(1);
}
```

3. Finally, we close the client pipe:

```
void end_resp_to_client(void)
{
    #if DEBUG_TRACE
        printf("%d :- end_resp_to_client()\n",  getpid());
    #endif

    if (client_fd != -1) {
        (void)close(client_fd);
        client_fd = -1;
    }
}
```

Client-side Functions

Complementing the server are the client functions in **pipe_imp.c**. Very similar, except for the worryingly-name **send_mess_to_server**.

Try It Out - Client Functions

1. After checking that a server is accessible, the **client_ starting** function initiallizes the client-side pipe:

```
int client_starting(void)
{
    #if DEBUG_TRACE
        printf("%d :- client_starting\n",  getpid());
    #endif

    mypid = getpid();
    if ((server_fd = open(SERVER_PIPE, O_WRONLY)) == -1) {
        fprintf(stderr, "Server not running\n");
        return(0);
    }

    (void)sprintf(client_pipe_name, CLIENT_PIPE, mypid);
    (void)unlink(client_pipe_name);
    if (mkfifo(client_pipe_name, 0777) == -1) {
        fprintf(stderr, "Unable to create client pipe %s\n",
                    client_pipe_name);
        return(0);
    }
    return(1);
}
```

2. The **client_ending** function closes file descriptors and deletes the now redundant named pipe:

```
void client_ending(void)
{
    #if DEBUG_TRACE
        printf("%d :- client_ending()\n",  getpid());
    #endif

    if (client_write_fd != -1) (void)close(client_write_fd);
    if (client_fd != -1) (void)close(client_fd);
    if (server_fd != -1) (void)close(server_fd);
    (void)unlink(client_pipe_name);
}
```

3. The **send_mess_to_server** function passes the request through the server pipe:

```
int send_mess_to_server(message_db_t mess_to_send)
{
    int write_bytes;

    #if DEBUG_TRACE
        printf("%d :- send_mess_to_server()\n",  getpid());
    #endif

    if (server_fd == -1) return(0);
    mess_to_send.client_pid = mypid;
    write_bytes = write(server_fd, &mess_to_send, sizeof(mess_to_send));
```

```
    if (write_bytes != sizeof(mess_to_send)) return(0)
    return(1);
}
```

Try It Out - Getting Server Results

1. This client function starts to listen for the server response.

```
int start_resp_from_server(void)
{
    #if DEBUG_TRACE
        printf("%d :- start_resp_from_server()\n",  getpid());
    #endif

    if (client_pipe_name[0] == '\0') return(0);
    if (client_fd != -1) return(1);

    client_fd = open(client_pipe_name, O_RDONLY);
    if (client_fd != -1) {
        client_write_fd = open(client_pipe_name, O_WRONLY);
        if (client_write_fd != -1) return(1);
        (void)close(client_fd);
        client_fd = -1;
    }
    return(0);
}
```

2. Here's the main **read** from the server which gets the matching database entries:

```
int read_resp_from_server(message_db_t *rec_ptr)
{
    int read_bytes;
    int return_code = 0;

    #if DEBUG_TRACE
        printf("%d :- read_resp_from_server()\n",  getpid());
    #endif

    if (!rec_ptr) return(0);
    if (client_fd == -1) return(0);

    read_bytes = read(client_fd, rec_ptr, sizeof(*rec_ptr));
    if (read_bytes == sizeof(*rec_ptr)) return_code = 1;
    return(return_code);
}
```

3. And finally, the client function that marks the end of the server response.

```
void end_resp_from_server(void)
{
    #if DEBUG_TRACE
        printf("%d :- end_resp_from_server()\n",  getpid());
    #endif

    /* This function is empty in the pipe implementation */
}
```

How It Works

The second, additional **open** of the client pipe for writing in **start_resp_from_server**,

```
client_write_fd = open(client_pipe_name, O_WRONLY);
```

is used to prevent a race condition.

Finally, here's the makefile to put it altogether:

```
all: server_app client_app

CC=gcc
CFLAGS=-pedantic -Wall

# For debugging un-comment the next line
# DFLAGS=-DDEBUG_TRACE=1 -g

.c.o:
	$(CC) $(CFLAGS) $(DFLAGS) -c $<
```

```
app_ui.o: app_ui.c cd_data.h
cd_dbm.o: cd_dbm.c cd_data.h
client_if.o: client_if.c cd_data.h cliserv.h
pipe_imp.o: pipe_imp.c cd_data.h cliserv.h
server.o: server.c cd_data.h cliserv.h


client_app: app_ui.o client_if.o pipe_imp.o
	$(CC) -o client_app  $(DFLAGS) app_ui.o client_if.o pipe_imp.o

server_app:	server.o cd_dbm.o pipe_imp.o
	$(CC) -o server_app  $(DFLAGS) server.o cd_dbm.o pipe_imp.o -ldbm
```

Application Summary

We've now separated our CD database application into a client and a server.

Summary

This capter looked at passing data between processes using pipes.

CS 248 - UNIX Programming Web Site Menu
Information | Syllabus | Schedule | Online "Lectures" | Projects | Quizzes | Web Board

---