Programmation Systèmes
Cours 6 — IPC: FIFO

Stefano Zacchiroli
zack@pps.univ-paris-diderot.fr

Laboratoire PPS, Université Paris Diderot

2012–2013

Outline

1. FIFOs
2. FIFO-based architectures

Outline

1. FIFOs
2. FIFO-based architectures

Pipes — looking back

Pros:

1. very simple data transfer model with implicit synchronization on read/write operations
2. use well-known and versatile handles: file descriptors
3. simple access control model: create before fork, all related processes can access
4. highly portable

Cons:

1. can be used only by related processes
2. communication is kernel mediated and requires double copy

For most applications, (2) is not a big deal.
On the other hand, (1) makes impossible quite a number of architectures (e.g. client-server among unrelated processes).

On pipes restrictions

> *Why can't pipe be used for communication across unrelated processes?*

[1] we can pass FDs around via UNIX domain sockets, but then we already have an IPC mechanism among unrelated processes.

On pipes restrictions

Why can't pipe be used for communication across unrelated processes?

1. naming scheme ⟶ pipes are anonymous
   - they are requested to the kernel and accessed via FDs
   - there is no (handy) way to reference them from "the outside"[1]
2. access control
   - "all or nothing" among related processes who see the FD
   - ... and "all or nothing" is too coarse-grained for *unrelated* processes

[1] we can pass FDs around via UNIX domain sockets, but then we already have an IPC mechanism among unrelated processes...

Named pipes

To overcome pipes limitations we need:

- a naming scheme that is valid between unrelated processes
    -
- a fine-grained access control mechanism
    -

Named pipes

To overcome pipes limitations we need:

- a naming scheme that is valid between unrelated processes
  - idea: let's use filesystem pathnames
- a fine-grained access control mechanism
  - idea: let's use filesystem permission masks

Let's use the filesystem!

Design choice coherent with UNIX "*everything is a file*" mantra.

Putting the pieces together we obtain FIFOs, AKA named pipes:

- conceptually similar to pipes,
- but exist on the filesystem and are accessed through it.

FIFOs — IPC characteristics

- data transfer, byte stream IPC facility that connect processes; the byte stream written to one end of the FIFO can be read from the other
- pathname identifiers are used to rendez-vous on FIFOs
- once opened, FIFOs are referenced by file descriptor handles
- accessible by unrelated processes
- filesystem-persistent; they disappear when unlinked (i.e. "deleted") from the containing directory
- highly portable: available on all known UNIX-es

FIFOs — file system details

- FIFOs can be created in the shell using `mkfifo(1)`
- a FIFO is a special file (i.e. non regular) that exists on the file system
  - `S_ISFIFO()` will return true on `stat`'s `s_mode` field
  - `lseek` will fail and set `errno` to ESPIPE (the same happen for pipes and sockets)
- permission masks can be decided upon creation and/or changed with chmod, as usual (thanks to uniformity)
- FIFOs can be used by programs as ordinary files
  - `open`, `read`, `write`, `unlink`, etc.
  - . . . as long as seeking is not needed

FIFOs — file system details

- FIFOs can be created in the shell using `mkfifo(1)`
- a FIFO is a special file (i.e. non regular) that exists on the file system
  - `S_ISFIFO()` will return true on `stat`'s `s_mode` field
  - `lseek` will fail and set `errno` to `ESPIPE` (the same happen for pipes and sockets)
- permission masks can be decided upon creation and/or changed with chmod, as usual (thanks to uniformity)
- FIFOs can be used by programs as ordinary files
  - `open`, `read`, `write`, `unlink`, etc.
  - . . . as long as seeking is not needed

Demo

FIFO creation

Given that open and `creat` only allow to create regular files, we need a different system call to create FIFOs.
`mkfifo` (homonymous with the command line utility) allows to do so:

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

int mkfifo(const char *pathname, mode_t mode);
```

Returns: *0 if OK, -1 on error*

- mode has the same semantics of open/`creat` syscalls: it specifies the desired permission mask for the FIFO
  - it will be bitwise AND-ed with the complement of current umask
- on most UNIX implementations, `mkfifo` is a wrapper around the more generic mknod, that allows to create all kinds of files
  - `mkfifo(path) = mknod(path, S_IFIFO, 0)`
  - the above is in fact the *only* portable use of `mknod`

FIFO — example

```c
#include <errno.h>
#include <fcntl.h>
#include <sys/stat.h>
#include <unistd.h>
#include "helpers.h"

#define BUFFSIZE        4096
#define FIFO_PATH       "fifo"
int main(void) {
	int n, fd;
	char buf[BUFFSIZE];

	if (mkfifo(FIFO_PATH, S_IRUSR | S_IWUSR) < 0
	    && errno != EEXIST)
		err_sys("fifo error");
	printf("opening %s...\n", FIFO_PATH);
	if ((fd = open(FIFO_PATH, O_RDONLY)) < 0)
		err_sys("open error");
	printf("entering main loop...\n");
	while ((n = read(fd, buf, BUFFSIZE)) > 0)
		if (write(STDOUT_FILENO, buf, n) != n)
			err_sys("write error");
	if (n < 0)  err_sys("read error");
	exit(EXIT_SUCCESS);
} // fifo-cat.c
```

FIFO — example (cont.)

Demo

Notes:

- we create the FIFO only if needed, otherwise we reuse the existing one (filesystem persistence)
- open blocks until a writer arrives
- when the *last* writer terminates, the reader gets a EOF

FIFOs — synchronization

The most simple use case for FIFOs is synchronization between 2 processes: 1 reading from + 1 writing to the FIFO.

FIFOs have an unusual open semantics built around it:

- a process opening a FIFO for reading (O_RDONLY) will block. . .
  - note: usually read could block waiting for data, open could not
- a process opening a FIFO for writing (O_WRONLY) will block. . .
- . . . until another process opens the FIFO for the complementary action

The kernel enforces 2-peer synchronization upon FIFO open.

- if the other end of the FIFO is already open (i.e. after synchronization between 2 processes has already happened), open will return immediately

FIFOs — non-blocking behavior

In those rare cases when you want to avoid blocking on FIFOs open, you can request non-blocking I/O with the O_NONBLOCK open flag.

- open with O_RDONLY will return immediately (opening read end)
  - read-s on the FIFO before any connected writer will become available won't block, but rather return no data (coherently with the usual discipline of non-blocking I/O)
- open with O_WRONLY will return an ENXIO error until the read end of the FIFO has already been opened

They behave differently because the consequences of writing to a FIFO with no readers are more severe (SIGPIPE).

Table: FIFO open behavior

type	flags	other end open	other end closed
reading	no flags	immediate success	blocks
reading	O_NONBLOCK	immediate success	immediate success
writing	no flags	immediate success	blocks
writing	O_NONBLOCK	immediate success	fails (ENXIO)

FIFOs — the O_RDWR trick

On many UNIX-es (including Linux), opening a FIFO with the O_RDWR will never block. It will return successfully and mark both ends as open.

- used to open a FIFO for writing before a reader is available
- non portable, use with caution
  - or prefer O_NONBLOCK all together

FIFOs — multiple readers/writers

Multiple readers/writer to FIFOs are common.

The main intuition to keep in mind is that the kernel maintains a count of the number of connected readers/writers and will not "complain" until at least one reader and one writer exist.

- as it happens with pipes, writing to a FIFO with no connected readers will fail with EPIPE and also deliver a SIGPIPE signal to the writer
- reading to a FIFO with no connected writers will return an EOF to the reader (as we've seen)

The O_RDWR trick can be used to fool the count of connected readers/writers and ensure that both are above zero.

O_RDWR trick — example

```
#include <errno.h>
#include <fcntl.h>
#include <sys/stat.h>
#include <unistd.h>
#include "helpers.h"

#define BUFFSIZE        4096
#define FIFO_PATH       "fifo"
int main(void) {
        int n, fd;
        char buf[BUFFSIZE];

        if (mkfifo(FIFO_PATH, S_IRUSR | S_IWUSR) < 0
            && errno != EEXIST)
                err_sys("fifo error");
        printf("opening %s...\n", FIFO_PATH);
        if ((fd = open(FIFO_PATH, O_RDWR)) < 0) /* non portable! */
                err_sys("open error");
        printf("entering main loop...\n");
        while ((n = read(fd, buf, BUFFSIZE)) > 0)
                if (write(STDOUT_FILENO, buf, n) != n)
                        err_sys("write error");
        if (n < 0)  err_sys("read error");
        exit(EXIT_SUCCESS);
} // fifo-cat-trick.c
```

Demo

Notes:

- the only difference is in the O_RDWR flag
- open no longer blocks
- the program is now persistent: it will not die when the last writer disappear and can serve subsequent writers

(what would happen if we connect multiple reader to the same pipe?)

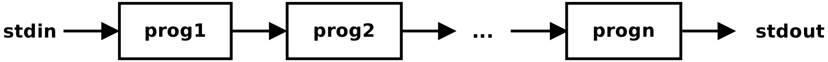
Outline

1 FIFOs

2 FIFO-based architectures

Linear process combination

Shell pipelines $p_1 \mid \ldots \mid p_n$ allow to create linear combinations where n processes (usually *filters*) cooperate using $n - 1$ UNIX pipes (created by the shell).



- the input of each process is either STDIN or the output of the previous process in the pipeline
- the output of each process is either STDOUT or the output of the next process in the pipeline
- the output of each process is consumed exactly once

Relying only on pipelines, we cannot process the output of a given process more than once (i.e. non-linearly).

Duplicating output streams with FIFOs

Use case

We want to process STDIN with a first filter prog1 and then process its output with two programs—prog2 and prog3—without using temporary files.

Duplicating output streams with FIFOs

Use case

We want to process STDIN with a first filter prog1 and then process its output with two programs—prog2 and prog3—without using temporary files.

We will use two tools:

1. tee(1) — a cat replacement that also writes to a file
   - mnemonic: “tee” as the letter “T”
2. FIFOs

Duplicating output streams with FIFOs

Use case

We want to process STDIN with a first filter prog1 and then process its output with two programs—prog2 and prog3—without using temporary files.

We will use two tools:

1. tee(1) — a cat replacement that also writes to a file
   - mnemonic: “tee” as the letter “T”
2. FIFOs

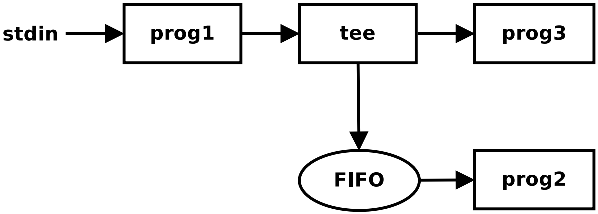


Figure: process arrangement

Duplicating output streams with FIFOs — example

```
$ mkfifo fifo
$ wc -l < fifo &
$ ls -l | tee fifo | sort -k5n
<snip>
$
```

Duplicating output streams with FIFOs — example

```
$ mkfifo fifo
$ wc -l < fifo &
$ ls -l | tee fifo | sort -k5n
<snip>
$
```

will show the number of (non-hidden) files in the current working directory (thanks to `wc -l`), as well as the details of each of them presented in increasing size order (thanks to `ls -l`)

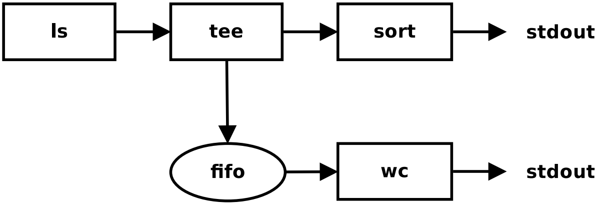


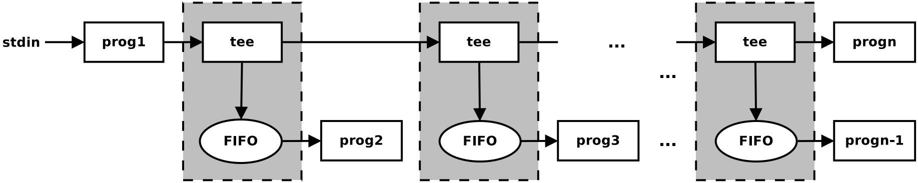
Figure: process arrangement

Multiplying output streams with FIFOs ?

can we generalize the scheme to *multiply* the output and process it *n* times?

Multiplying output streams with FIFOs

- intuition: each FIFO/tee block allows to add one extra branch in the pipeline
- we can scale up to n extra branches with $n-1$ FIFO/tee blocks



Client-server FIFOs

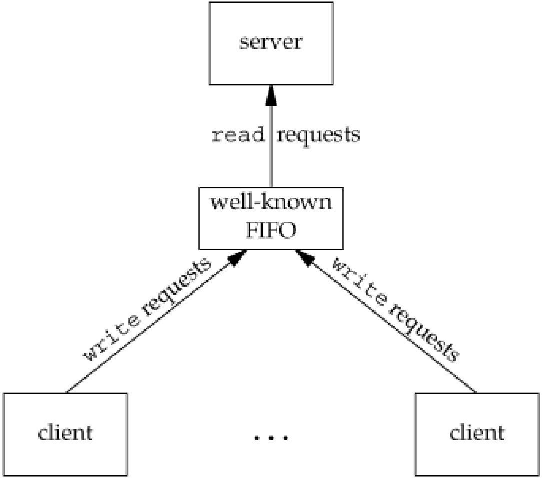
FIFOs are used as the basic communication device in client-server architectures meant to run on a single machine

- e.g. daemons offering system services to local programs
- to find some, try `find /var -type p` (as root)

In its simplest form, a FIFO-based client-server architecture works as follows:

0. a filesystem path pointing to a FIFO is agreed upon by server/clients and used as the well-known address of the service
1. the FIFO is either persistent (e.g. created at install-time) or created by the server at startup
2. the clients write requests to the FIFO
3. the server reads requests from the FIFO and handles them sequentially

Client-server FIFOs — architecture



APUE, Figure 15.22

Client-server FIFOs — atomic `write`

Even with such a simple architecture, we need to worry about race conditions. For all requests of size greater than 1 byte, we need to worry about interleaving issues, since the FIFO is shared among the server and all its clients.

- the usual solution would be to do proper synchronization and ensure that only one client at a time write its request to the FIFO

Client-server FIFOs — atomic `write`

Even with such a simple architecture, we need to worry about race conditions. For all requests of size greater than 1 byte, we need to worry about interleaving issues, since the FIFO is shared among the server and all its clients.

- the usual solution would be to do proper synchronization and ensure that only one client at a time write its request to the FIFO
- luckily, UNIX kernels offer a cleaner solution

> All `write` of size PIPE_BUF or less to pipes or FIFOs are guaranteed (by POSIX) to be atomic.

- if all `write`-s to the shared FIFO are smaller than PIPE_BUF, no interleaving can happen

How much is PIPE_BUF?

The value is implementation dependent, but with a guaranteed minimum of 512 bytes.

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

int main(void) {
        printf("PIPE_BUF: %d\n", PIPE_BUF);
        exit(EXIT_SUCCESS);
}
```

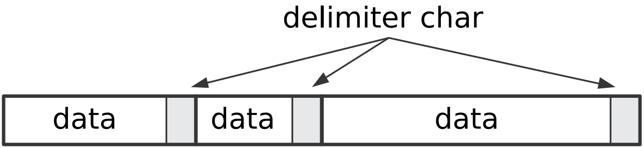
```
$ ./pipe-buf
PIPE_BUF: 4096          # on Linux x86, 64 bits
$
```

It's more than enough for most *control* languages, but we need to use different IPC objects for actual atomic *data* transfer.

Messages within byte streams

FIFOs offer a byte stream IPC facility, while client-server architectures often rely on separate messages. There are several ways to do message-oriented communication with byte streams:

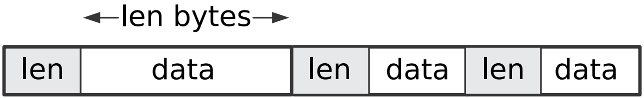
1. terminate each message with a delimiter character
   - pro: easy for the sender
   - con: *might* need to escape the delimiter character
   - con: forces receiver to scan the IPC object one char at a time



Messages within byte streams (cont.)

FIFOs offer a byte stream IPC facility, while client-server architectures often rely on separate messages. There are several ways to do message-oriented communication with byte streams:

2. prefix each message with a fixed-size header containing a length field
    - pro: efficient (read the header first, than the rest)
    - con: malformed messages (hence the need of CRC or equiv.)



Messages within byte streams (cont.)

FIFOs offer a byte stream IPC facility, while client-server architectures often rely on separate messages. There are several ways to do message-oriented communication with byte streams:

3. use fixed-length messages
    - pro: very simple to program
    - con: impose a maximum message size
    - con: padding

Messages within byte streams (cont.)

FIFOs offer a byte stream IPC facility, while client-server architectures often rely on separate messages. There are several ways to do message-oriented communication with byte streams:

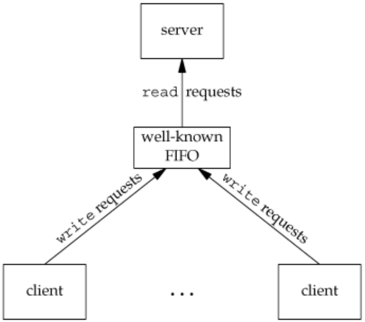
In the case of pipes and FIFOs, we *additionally* have PIPE_BUF as message size upper bound to avoid interleaving.

Client-server FIFOs — example

Example

We want to implement a local server that, upon reading a “reload” command from a FIFO, will reread the `/etc/motd` file from disk and print it on stdout.

- using the architecture we have discussed. . .
- . . . and fixed-length messages



Client-server FIFOs — example (protocol)

```
#include <errno.h>
#include <fcntl.h>
#include <string.h>
#include <sys/stat.h>
#include <unistd.h>
#include "helpers.h"

#define FIFO_PATH       "fifo"
#define ACT_RELOAD      17

struct request {
        int action;     /* one of ACT_* macros */
};

// common-1.h
```

Client-server FIFOs — example (server)

```
#include "common-1.h"

#define BUFFSIZE        4096
#define MOTD_PATH       "/etc/motd"

void print_motd(void) {
        int fd, n;
        char buf[BUFFSIZE];
        if ((fd = open(MOTD_PATH, O_RDONLY)) < 0)
                err_sys("open error");
        while ((n = read(fd, buf, BUFFSIZE)) > 0)
                if (write(STDOUT_FILENO, buf, n) != n)
                        err_sys("write error");
        close(fd);
}
```

Client-server FIFOs — example (server) (cont.)

```
int main(void) {
        int fd;
        struct request req;
        if (mkfifo(FIFO_PATH, S_IRUSR | S_IWUSR) < 0 && errno != EEXIST)
                err_sys("fifo error");
        if ((fd = open(FIFO_PATH, O_RDWR)) < 0)
                err_sys("open error");
        print_motd();
        for (;;) {
                if (read(fd, &req, sizeof(struct request))
                    != sizeof(struct request))
                        continue;       /* partial read or error*/
                switch (req.action) {
                case ACT_RELOAD:
                        printf("**** reload ****\n");
                        print_motd();
                        break;
                default:
                        printf("**** invalid request ****\n");
                        break;
                }
        }
        exit(EXIT_SUCCESS);
} // server-1.c
```

Client-server FIFOs — example (client)

```c
#include "common-1.h"

int main(void) {
	int fd;
	struct request req;

	if ((fd = open(FIFO_PATH, O_WRONLY)) < 0)
		err_sys("open error");

	req.action = ACT_RELOAD;
	if (write(fd, &req, sizeof(struct request))
	    != sizeof(struct request))
		err_sys("write error");

	exit(EXIT_SUCCESS);
} // client-1.c
```

Client-server FIFOs — example

Demo

Client-server FIFOs — request-response

The previous architecture is not suitable for client-server request-response schemes where the server, in response to incoming request, has both to act and reply to the client.

why?

Client-server FIFOs — request-response

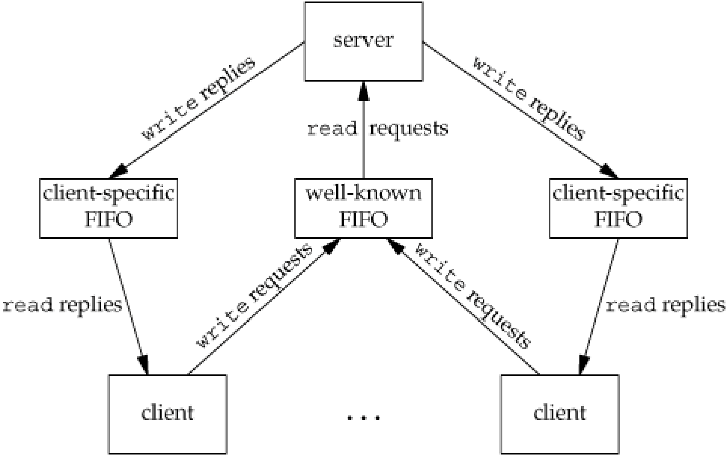
The previous architecture is not suitable for client-server request-response schemes where the server, in response to incoming request, has both to act and reply to the client.

The problem: we cannot send replies trough the shared FIFO, because we don't know which of the client will read the message. We need a context where to correlate responses with the corresponding requests.

To do so we can:

- use the shared FIFO for incoming requests only
- use client-specific FIFOs (one per client) to send back responses

Client-server request-response FIFOs — architecture



APUE, Figure 15.23

Client-server request-response FIFOs — naming

For the architecture to work, clients and server must agree on the pathname of the client-specific FIFO.

Common solutions are:

- the client tells the server where he should send the response, by including the pathname in the request
- clients and server agrees on a naming scheme based on some client identifier, and the client sends the identifier as part of the request
  - e.g. we say that client-specific FIFOs will be named

    `/var/run/my-server/client-%d.fifo`

    where %d is the client PID[2]

[2] we are not yet considering security issues here. . .

Request-response FIFOs — example

Example

We want to implement a server that allocates unique sequential identifiers to clients.

- the server hold a global integer counter
- the client connect to the server to request a new unique identifier in the sequence
- the server send back the next integer in the sequence and update the global counter

We will use a client-server request-response FIFO architecture:

- client → server requests are sent via a shared FIFO
- server → client responses are sent via client-specific FIFOs
- server → client rendez-vous happens via a PID-based naming scheme

Request-response FIFOs — example (protocol)

```c
#include <errno.h>
#include <fcntl.h>
#include <signal.h>
#include <string.h>
#include <sys/stat.h>
#include <unistd.h>
#include "helpers.h"

#define SRV_FIFO        "seqnum-srv"
#define CLI_FIFO_TPL    "seqnum-cli.%ld"
#define CLI_FIFO_LEN    (sizeof(CLI_FIFO_TPL) + 20)

struct request {        /* Request (client --> server) */
        pid_t pid;      /* PID of client */
};

struct response {       /* Response (server --> client) */
        int seqno;      /* Sequence number */
};

/* fifo_seqnum.h --- based on TLPI's fifo_seqnum.h
   Copyright (C) Michael Kerrisk, 2010. License: GNU AGPL */
```

Request-response FIFOs — example (server)

```c
#include "fifo_seqnum.h"

int main(void) {
	int srv_fd, cli_fd;
	char cli_fifo[CLI_FIFO_LEN];
	struct request req;
	struct response res;
	int seqno = 0;

	if (mkfifo(SRV_FIFO, S_IRUSR | S_IWUSR | S_IWGRP) < 0
	    && errno != EEXIST)
		err_sys("mkfifo error");
	if ((srv_fd = open(SRV_FIFO, O_RDWR)) < 0)
		err_sys("open error");
	if (signal(SIGPIPE, SIG_IGN) == SIG_ERR)
		err_sys("signal");

	for (;;) {		/* main request/response loop */
		if (read(srv_fd, &req, sizeof(struct request))
		    != sizeof(struct request))
			continue;
```

Request-response FIFOs — example (server) (cont.)

```
                snprintf(cli_fifo, CLI_FIFO_LEN, CLI_FIFO_TPL,
                         (long) req.pid);
                if ((cli_fd = open(cli_fifo, O_WRONLY)) < 0) {
                        err_msg("open error (client FIFO)");
                        continue;
                }

                res.seqno = seqno;
                if (write(cli_fd, &res, sizeof(struct response))
                    != sizeof(struct response))
                        err_msg("write error (client FIFO)");
                if (close(cli_fd) == -1)
                        err_msg("close");

                seqno++;
        }
}

/* fifo_seqnum_server.c --- based on TLPI's fifo_seqnum_server.c
   Copyright (C) Michael Kerrisk, 2010. License: GNU AGPL */
```

Request-response FIFOs — example (client)

```
#include "fifo_seqnum.h"

static char cli_fifo[CLI_FIFO_LEN];

void remove_fifo(void) {
        unlink(cli_fifo);
}

int main(void) {
    int srv_fd, cli_fd;
    struct request req;
    struct response resp;

    snprintf(cli_fifo, CLI_FIFO_LEN, CLI_FIFO_TPL, (long) getpid());
    if (mkfifo(cli_fifo, S_IRUSR | S_IWUSR | S_IWGRP) == -1
        && errno != EEXIST)
            err_sys("mkfifo error");
    if (atexit(remove_fifo) != 0)
            err_sys("atexit error");
```

Request-response FIFOs — example (client) (cont.)

```
    req.pid = getpid();
    if ((srv_fd = open(SRV_FIFO, O_WRONLY)) < 0)
            err_sys("open error (server FIFO)");
    if (write(srv_fd, &req, sizeof(struct request)) !=
        sizeof(struct request))
            err_sys("write error");

    if ((cli_fd = open(cli_fifo, O_RDONLY)) < 0)
            err_sys("open error (client FIFO)");

    if (read(cli_fd, &resp, sizeof(struct response))
        != sizeof(struct response))
            err_sys("read error");

    printf("%d\n", resp.seqno);
    exit(EXIT_SUCCESS);
}

/* fifo_seqnum_client.c --- based on TLPI's fifo_seqnum_client.c
   Copyright (C) Michael Kerrisk, 2010. License: GNU AGPL */
```

Demo