UNIX Systems Programming

Interprocess Communication

(Curry, chp.13)

Dr. Kivanç Dinçer

CENG-332 Lecture Notes

Spring 2000

Pipes

- Two processes can communicate with each other by exchanging data
- Pipes is a special pair of file descriptors that, rather than being connected to a file, is connected to another process.
  - provides an interface bw two processes
  - provides a unidirectional communications medium

Creating Pipeline Commands

- Once a pipe is created, there is very little difference bw a pipe file descriptor and a regular file descriptor.

```
% eqn report > out1
% tbl out1 > out2
% troff out2 > out3
% psdit out3 > out4
% lp out4
% rm out1 out2 out3 out4
```

`eqn report | tbl | troff | psdit | lp`

A filter is a program that will read from its standard input and write to its standard output.

  - programs written in this way can be joined together in pipelines by the shell.

Single Pipe Creation

```
#include <stdio.h>
FILE *fopen(const char *command, const char *type);
```

where `type` is `r` (open the file for reading) or `w` (for writing)

returns NULL if error occurs

  - creates a new process and executes the command.
  - creates a pipe to that process and connects it to process' stdin or stdout.
  - returns a file pointer to the calling process.

```
#include <stdio.h>
int *fclose(FILE *stream);
```

  - closes the stream and frees up the buffers associated with it.
  - also issues a call to `waitpid` to wait for the child process to terminate, then returns child's termination status to the caller.

See Example 13-1: popen is quite inefficient (it starts a copy of the shell,) system calls and library routines are more efficient than using popen.

Advanced Pipe Creation

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

- returns 0
- return –1 if failure and places the reason for failure in errno.
- creates two file descriptors:
  - fd[0] is open for reading
  - fd[1] is open for writing
  - The two file descriptors are joined like a pipe, such that data written to fd[1] can be read from fd[0].

- After creating a pipe, the calling process normally calls fork to create a child process.
  - The two processes can then communicate, in one direction, using the pipe.
  - A pipe is a half-duplex communications channel.

Closing a Pipe

- If the write end of a pipe has been closed,
  - any further reads from the pipe will return 0, or end-of-file.
- If the read end of a pipe has been closed,
  - any attempt to write to the pipe will result in a SIGPIPE signal being delivered to the process attempting the write.

Each pipe has a buffer size, this size is defined by the constant PIPE_BUF, in limits.h.

- A write of this many bytes or less is guaranteed not to be interleaved with the writes from other processes writing the same pipe.

See Example 13-2: pipedate: we create a pipe and execute date ourselves.

See Example 13-3: pipemail: uses the pipe for the parent to send data to the child.

FIFOs (Named Pipes)

- Major limitation of pipes:
  - they can only be used bw related processes

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

The mode argument contains a set of permission bits to set the FIFO returns 0 or –1 and sets errno if fails.

Opening a FIFO:

- default: O_NONBLOCK not specified: an open for reading/writing only blocks until another process opens the FIFO for writing/reading.
- O_NONBLOCK is specified: an open for reading returns immediately, an open for writing returns an error if FIFO has not been yet opened for reading.

See Example 13-4 & 13-5: a server and a client using FIFOs to communicate, server prints any data it receives from the client.

UNIX-Domain Sockets

vs. named pipes:

- similar in providing an address in the file system that unrelated processes may use to communicate
- FIFOs are accessed just like any other file. UNIX-domain sockets are implemented using the Berkeley networking paradigm, usually called the socket interface (create, destroy, transfer, … functions)

IPC with sockets follow the Client-Server Model:

- The server is responsible for satisfying the requests made of it by other processes, called clients.
  - a server usually has a well-known address.

Creating a Socket

```
#include <sys/types.h>
#include <sys/socket.h>
int socket(int domain, int type, int protocol)
```

- domain specifies the domain, or address family, in which addresses should be interpreted.
  - It imposes certain restrictions on the length of addresses, and what they mean.
  - AF_UNIX domain is used for UNIX-domain sockets.
- protocol specifies the protocol number that should be used on the socket, usually the same as address family.
  - PF_UNIX protocol is used for UNIX-domain sockets.

- type specifies the communications channels supported by sockets:
  - SOCK_STREAM (virtual circuit)
    - a bi-directional continuous byte stream that guarantees the reliable delivery of data in the order in which it was sent.The circuit remains intact until the conversation is complete.
  - SOCK_DGRAM
    - used to send distinct packets of info called datagrams. No guarantees on order or delivery.

> returns a socket descriptor (a non-negative integer similar to a fd) or –1 and errno.

```
#include <sys/types.h>
#include <sys/socket.h>
int socketpair(int domain, int type, int protocol,int  sv[2])
```

creates an unamed pair of sockets and placed their descriptors in sd. Each socket is a bidirectional communications channel.

returns 0 or –1 and errno.

Server-Side Functions: bind, listen, accept

1- Naming a socket

- A server process must provide a socket with a name, so that client programs can access it.

```
#include <sys/types.h>
#include <sys/socket.h>
int bind(int s, const struct sockaddr* name, int addlen)
```

Note: address must not be already in use!

```
struct sockaddr_un {
       short sun_family;      <-----  always AF_UNIX
       char  sun_path[108];   <-----  system pathname of socket
};
```

Note: file should not already exist!

Server-Side Functions: bind, listen, accept

2- Waiting for Connections

- Server must notify the O/S when it is ready to accept connections from clients on that socket.

```
int listen(int s, int backlog)
```

backlog specifies the # of connection requests that may be pending at any given time.

3- Accepting Connections

```
int accept(int s, struct sockaddr *name, int *addrlen)
```

- returns a new sd to communicate with the client.
  - old sd continue to accept additional connections.
- When connection is accepted, if name is not null, O/S stores the address of the client there and length in addrlen.

returns –1 and errno if fails.

Client-Side: Connecting to a Server

```
int connect(int s, struct sockaddr *name, int addrlen)
```

- connects the socket ref'd by s to the server at addr described by name.
- addrlen specifies the length of addr in name.

returns 0 or –1 and errno.

A client may use connect to connect to a datagram socket to the server as well.

- Not necessary
- But it does enable the client to send datagrams on the socket w/o having to specify destination addr for each datagram.

Client-Side: Transferring Data

1- simply use read and write.

2- use send and recv

```
int recv(int s, char *buf,       int len, int flags)
int send(int s, const char *buf, int len, int flags)
```

flag effect how the data is sent or received.

MSG_PEEK: If specified in a call to recv, the data is copied into buf as usual, but it is not consumed. Another call to recv will return the same data.

Transferring Data using Datagram-Based Sockets

- client does not (generally)call connect
  - There is no way for the O/S to determine automatically where data on these sockets is to be sent.
- server does not call listen or accept

The sender must tell the O/S each time where the data is to be delivered, and the receiver must ask where it came from.

```
int recvfrom(int s, char *buf, int len, int flags,
        struct sockaddr *from, int *fromlen)
int send(int s, const char *buf, int len, int flags,
        struct sockaddr *to,   int tolen)
```

returns # bytes actually received/sent or –1.

Destroying the Communications Channel

1- Close a socket with the close function

- if the socket refers to a stream-based socket, the close will block until all data has been transmitted.

2- Use shutdown function

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

shuts down either or both sides of the communications channel

- how is 0: shut down for reading, all further reads from the socket return eof.
- how is 1: shut down for writing, all further writes to the socket will fail.
- how is 2: shut down both sides