MONASH University

FIT2100 Laboratory #6
Interprocess Communication in Unix/Linux
Week 12 Semester 2 2024

August 16, 2024

Revision Status:

`Updated by Dr.Charith Jayasekara, Aug 2024.`

The contents presented in this Laboratory (including the Laboratory tasks) were adapted from David Curry's texts by Dr Jojo Wong.

- David A. Curry (1989). *C on the UNIX System*, O'Reilly.
- David A. Curry (1996). *UNIX Systems Programming for SVR4*, O'Reilly.

Contents

1 Background

This Laboratory is aimed to extend your knowledge on the concepts of interprocess communication in the Unix/Linux environment. We will explore four different mechanisms which enable two processes executing on the same computer system to communicate with each other: *basic pipes*, *named pipes (FIFO)*, *message queues*, and *sockets*.

Before you attempt any of the tasks in this LAB, create a folder named `LAB06` under the FIT2100 folder (`~/Document/FIT2100`). Save all your source files under this `LAB06` folder.

Before attending the Laboratory, you should:

- Complete your pre-class readings (Section 2)
- Attempt the Assessed Laboratory tasks (Section 3)

2 Pre-class Reading

2.1 Pipes

A *pipe* is the most basic form of interprocess communication that can be used to join two processes together. It is setup by a special pair of file descriptors which connect the two processes in communication.

When Process A writes data to its pipe file descriptor, Process B can read that data from its pipe file descriptor. Likewise, when Process B writes to its pipe file descriptor, Process A can then read the data from its pipe file descriptor.

In the Unix shell, you would have used the *pipeline* command (`|`). For example, the following Unix command sends the output of `ls` to `wc -l` to indicate the number of files or directories found in the current directory.[1]

```
1    $ ls | wc -l
```

[1] The `wc -l` command prints the number of lines read from standard input. In this case we are piping the standard output from `ls` to the standard input of the `wc` utility.

2.1.1 Creating a pipe

At a deeper level, a pipe is created with the system call `pipe()`. The `pipe()` function is defined under the `<unistd.h>` library. The function should be passed an integer array of size 2 for it to place two file descriptors into. The function returns `0` if a pipe is successfully created; it returns `-1` otherwise indicating a pipe cannot be created, and the reason for failure is stored in the external variable `errno`.

Function prototype for the `pipe()` function:

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

By invoking the `pipe()` function, two file descriptors are created:

- `pipefd[0]` is open for *reading*
- `pipefd[1]` is open for *writing*

Basically, the two file descriptors are connected as a pipe — which allows data written at the write end of the pipe (`pipefd[1]`) to be readable from the read end of the pipe (`pipefd[0]`).

After creating a pipe, the calling process (who makes the call to `pipe()`) usually creates a child process by calling `fork()`. The two related processes can then communicate using the pipe in *one* direction. Thus, either the parent may send data to the child or the child may send data to the parent, **but not in both directions**. If bi-directional communication is required, *two* pipes must be set up — one for the parent to use to send data to the child; and one for the child to use to send data to the parent.

Note: Each pipe has a limited buffer size described by the constant `PIPE_BUF` defined under the `<limits.h>` library. The size limit may differ from version to version of Unix implementations; for Linux, the minimum capacity is `4096` bytes. Also, it is possible to have more than one process writing to a pipe by applying the function `dup()` or `dup2()` (from `<unistd.h>`) on the file descriptor.

2.1.2 Closing the pipe

Communication can take place as long as both the read and write ends of a pipe are open. If one end of a pipe is closed, that would affect the communication between the two processes:

- If the read end of a pipe has been closed, any attempt to write to the pipe will result in a `SIGPIPE` signal being sent to the process attempting to write.
- If the write end of the pipe has been closed, any further reads from the pipe will return `0` as an indicator of the *end-of-file*.

Try using unnamed pipes yourself:

Download the `pipebox.c` program and run it.[2] Describe what the program does by adding a comment at the beginning of the source file.

2.2 Named Pipes (FIFO)

The un-named pipes introduced in the previous section work just like files. They are associated with file descriptors and are accessed by the low-level I/O functions — `read()` and `write()`. However, pipes do not exist in the file system and hence they do not have filenames or path names. One major limitation of pipes is that they can only be used between *related* processes (parent and child pairs).

Named pipes, also known as *FIFO special files*, are a variation that does associate with an entry in the file system. With a name attached, named pipes can then be used by processes that are *unrelated* to each other (not parent and child pairs), for both reading and writing.

2.2.1 Creating a named pipe

To create a named pipe, the built-in function `mkfifo()` is used.

Function prototype for the `mkfifo()` function:

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

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

Upon successful creation of a named pipe, `0` is returned. In the case of a failure, `-1` is returned (and `errno` is set accordingly to indicate the error).

The first parameter `pathname` specifies the path name of the named pipe to be created, where the path name must not already exist. The second parameter `mode` indicates the set of file permissions for the named pipe. For example, the following statement creates a named pipe "`/etc/mypipe`" that is readable and writable by all users.

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

mkfifo("/etc/mypipe", 0666);
```

[2] The warning message about missing fonts may be safely ignored.

2.2.2 Using a named pipe

Once a named pipe (FIFO) has been created, it must be opened for use with the `open()` function which we have been using for opening or creating a file for reading and/or writing (recall the tasks that we have done in Laboratory 2). Once the named pipe has been opened, the pipe can be accessed for reading and writing with the low-level I/O functions — `read()` and `write()` .

Just like unnamed pipes, a process that attempts to read from an empty FIFO will cause that process to become blocked and wait for a write. An attempt to write to a FIFO that has no processes reading it will result in a `SIGPIPE` signal. When the last writer on a FIFO closes it, the reader will receive an *end-of-file* indication.

Try using named pipes yourself:

Download the code for the `fifo_server.c` and `fifo_client.c` programs. These two programs deploy a FIFO (named pipe) for communication as a *server* and a *client*.

First, try to understand what the server program `fifo_server.c` does, and then complete the partial code presented for the client program `fifo_client.c`.

The server program creates a FIFO for reading and displays anything it receives from the named pipe to the standard output. The client program, on the other hand, opens the FIFO created by the server program for writing, and writes anything that it reads from the standard input to the named pipe.

The clearest way to observe the interaction between client and server, is to run each program in a *different* terminal window. In the first window, start the server:

```
1    $ ./fifo_server
```

...in the second terminal, start the client. Here we are redirecting the client's standard input from a file. If you wish to type standard input on the keyboard instead, note that the terminal uses `Ctrl-D` to signal end-of-file.

```
1    $ ./fifo_client < /etc/passwd
```

2.3 Message Queues

Message queues is another mechanism for interprocess communication in Unix. A message queue is represented as a linked list of message "packets" exchanged between processes, where each of a fixed maximum size. To preserve the order in which messages arrive, messages are added to the end of the queue. However, messages can be received in any order determined by the receiving processes based on the message type.

2.3.1 Setting up a message queue

A message queue is described as a structure of type `struct msqid_ds` defined in the `<sys/msg.h>` library and each queue is defined by a unique identifier (*queue id*). Before a process attempts to use a message queue, it must obtain the queue identifier by calling the `msgget()` function.

Function prototype for the `msgget()` function:

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>

int msgget(key_t key, int msgflg);
```

The first parameter `key` specifies the key to be associated with the message queue. If the key contains the value of zero (defined by the constant `IPC_PRIVATE`), a new message queue is always created. If the key contains a non-zero value, either a new queue is created, or the identifier of an existing queue is returned. This depends on whether the second parameter `msgflg` is set to the constant `IPC_CREAT` and whether the given `key` already exists.

If the `msgflg` parameter is set to both `IPC_CREAT` and `IPC_EXCL` and a message queue already exists with the given `key`, `-1` is returned indicating that the message queue cannot be created and the error that occurred is set in `errno`. However, if the creation is successful, the message queue identifier is returned.

2.3.2 Exchanging data with a message queue

Two functions are used for *sending* and *receiving* messages on a message queue — `msgsnd()` and `msgrcv()`.

Function prototypes for the `msgsnd()` and `msgrcv()` functions:

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>

int msgsnd(int msqid, const void *msgbuf, size_t msgsz, int msgflg);
int msgrcv(int msqid, void *msgbuf, size_t msgsz, long msgtype, int msgflg)
```

Sending data on a message queue

The `msgsnd()` function takes on four arguments: a message queue identifier (queue id), a reference to a user-defined message buffer, an integer indicating the size of the message, and a flag option. A message pointed by `msgbuf` (the second parameter) is sent on the message queue identified by `msqid` (the first parameter). If the message is successfully sent, the function return `0`; `-1` is returned otherwise highlighting an error has occurred.

Messages exchanged on a message queue can be defined using the following structure:

```
struct msgbuf {
    long    mtype;      /* message type, must be > 0 */
    char    mtext[];    /* message data */
}
```

The `mtype` field of the structure contains a positive integer value which can be used by the receiving process to select which type of message to receive. The `mtext` field represents the message buffer whose size is defined by `msgsz` in terms of bytes (the third parameter of the `msgsnd()` function).

Note: By default, if the message queue is full, the `msgsnd()` call will block until the queue becomes empty. If the `msgflg` parameter is set to `IPC_NOWAIT`, the `msgsnd()` call will fail with a failure code being returned.

Receiving data from a message queue

The `msgrcv()` function takes on five arguments: a message queue identifier (queue id), a reference to a user-defined message buffer, an integer indicating the size of the message received, an integer specifying the message type, and a flag option. If the message is successfully received, the number of bytes stored in the message buffer referenced by `msgbuf` is returned. If an error occurs, `-1` is returned and the error is set in `errno` accordingly.

When receiving messages, the receiving process must specify the message type (`msgtype`) in the `msgrcv()` function.

- If `msgtype` is set to the value of `0`, the first message in the queue will be read.
- If `msgtype` is set to a value greater than `0`, the first message in the queue of type `msgtype` will be read.
- If `msgtype` is set to a value less than `0`, the first message in the queue with the lowest type less than or equal to the absolute value of `msgtype` will be read.

Note: If no message of the requested type is available in the message queue and the parameter `msgflg` is set to `IPC_NOWAIT`, the `msgrcv()` call will fail with a failure code being returned. Otherwise, the `msgrcv()` call will block until a message of the specified type arrives in the message queue or the queue is removed from the system.

2.4 Sockets

Sockets are similar to named pipes in which they provide an address in the file system which unrelated processes may use for communication. However, sockets are different from named pipes in terms of how they are accessed. Sockets are implemented using the *socket interface* which consists of a set of functions to create, destroy and transfer data.

Interprocess communication with sockets is often described in terms of the client-server model which we have seen. The server program first requests the operating system (OS) for a socket. Once it is given with a socket, it then assigns a well-known name (address) to that socket. The name should always be the same, such that the client programs will know how to contact to the server program via that socket.

2.4.1 How does a socket work?

After a socket has been named, the server process listens on the socket for any connection requests from the client processes. When a connection request arrives, the server can choose to accept or reject. If the server accepted the connection, the OS connects the server and the client together at the socket.

The client process, on the other hand, begins by requesting for a socket from the OS. It then asks the OS to join its socket to a socket of a specific name that it would want to establish a connection. The OS attempts to search for a socket with the specific name. If such socket was found, the OS sends a connection request to the process which is listening on that socket (the server process).

Once the server and the client are connected at the socket, they can then communicate with each other by reading and writing data to and from the socket, just like a pipe.

2.4.2 Creating a socket

A socket is created using the built-in function `socket()` defined in the `<sys/socket.h>` library. A socket descriptor represented as a small non-negative integer (similar to a file descriptor) is returned when a socket is successfully created. If the function failed to create the socket, `-1` is returned and the reason for the failure is indicated in `errno`.

Function prototype for the `socket()` function:

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

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

The first parameter `domain` specifies the address family in which the address of the socket should be interpreted. For communication between processes on the same computer system, the `domain` parameter is assigned with the constant `AF_UNIX` (the Unix domain) — in which the addresses are treated as Unix path names.

The second parameter `type` defines which type of communication semantic (channel) that the socket supports. The two types which are of interest are `SOCK_STREAM` and `SOCK_DGRAM`. `SOCK_STREAM` supports a bi-directional communication with continuous byte streams — it is guaranteed that the data is delivered in the order it was sent, and no data can be sent until the connection is established. `SOCK_DGRAM` supports datagrams which are distinct packets of data — it is not guaranteed that the data is delivered in the order it was sent, and it is even not guaranteed that the data is delivered at all.

The last parameter `protocol` specifies a particular protocol to be used with the socket. The protocol number to use is specific to the communication domain (the address family). For the Unix domain (`AF_UNIX`), the corresponding protocol family is defined by the constant `PF_UNIX`. However, if the `protocol` parameter is set to `0`, the OS will determine which is the suitable protocol.

What does the server process need to do?

In order for the server process to exchange data with the client process, it has to invoke the following set of functions.

Naming a socket Once the socket has been created, the server process must provide a name (or a known address) to the socket by using the `bind()` function; otherwise the socket cannot be used by the client processes. If binding is successful, `0` is returned; if it fails to bind (often due to the address already being in use), `-1` is returned and the `errno` is set accordingly.

Function prototype for the `bind()` function:

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

int bind(int sockfd, const struct sockaddr *name, int addrlen);
```

Once the binding has been completed, the communication channel referenced by the first parameter `sockfd` is assigned with the address described by the second parameter `name`. The length of the address is indicated by the last parameter `addrlen`.

Note: The `name` parameter is a structure of a generic type (`sockaddr`) for socket addresses. In the Unix domain, the socket address is actually of type `sockaddr_un` with the following structure — where the `sun_family` represents the address family (set to `AF_UNIX`) and the `sun_path` represents the path name of the socket.

```
struct sockaddr_un {
    short sun_family;      /* AF_UNIX */
    char sun_path[108];      /* pathname */
}
```

Waiting for connections On a stream-based communication via a socket, the server process must notify the OS when it is ready to accept any connection requests from client processes. The function `listen()` is used for this purpose. On success, the function returns `0`; `-1` is returned otherwise.

Function prototype for the `listen()` function:

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

int listen(int sockfd, int backlog);
```

The socket that the server is listening on for connection requests is referenced by the first parameter `sockfd`. The second parameter `backlog` indicates the number of pending connection requests allowed. If a connection request arrives when the queue of pending connections is full, the client process will received an error indicating that the connection is refused.

Accepting connections When the server process has decided to accept the connection request, it uses the function `accept()` to achieve this. A new socket descriptor (a non-negative integer) will be returned, in which the server process can use this new socket descriptor to communicate with the client process. The existing socket descriptor from the server — the one with a known address assigned — is used for accepting other connection requests.

Function prototype for the `accept()` function:

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

int accept(int sockfd, const struct sockaddr *name, int *addrlen);
```

If the connection is accepted successfully, the socket address of the client process will be referenced by the second parameter `name` and its address length is stored in the last parameter `addrlen`. If unsuccessful, `-1` is returned and the reason for the failure is set in `errno`.

What does the client process need to do?

The client process, on the other hand, is required to connect to the server process that it intends to communicate on a stream-based socket (`SOCK_STREAM`). To do this, the client process should invoke the `connect()` function.

Function prototype for the `connect()` function:

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

int connect(int sockfd, const struct sockaddr *name, int addrlen);
```

The first parameter `sockfd` refers to a socket to be connected to the server process which the client intends to communicate with. The known address of the socket from the server end is referenced by the second parameter (`name`) where the length of the address is specified by the last parameter (`addrlen`). When the connection is successful, `0` is returned; `-1` is returned otherwise and the error is set in `errno` accordingly.

2.4.3 Transferring data with a socket

On a stream-based communication via a socket, the low-level I/O functions `read()` and `write()` can be used by the client and server processes. However, there are two functions specifically used with stream-based sockets — `recv()` and `send()`.

Function prototypes for the `recv()` and `send()` functions:

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

int recv(int sockfd, char *buffer, int len, int flags);
int send(int sockfd, const char *buffer, int len, int flags);
```

These two functions are in fact identical to `read()` and `write()`. However, there is a fourth argument which allows the program to specify options (described by *flags*) that would affect the way in which the data is sent or received.

One such option is defined by the constant `MSG_PEEK` — if this option is set in the first call to `recv()`, the data is copied to the `buffer` as usual but it is not "consumed" (the data is not yet read). The next call to `recv()` will return the same data. In a way, this enables a program to have a *peek* at the received data and can then decide what to do with the data without having to actually "read" it.

Destroying the socket The communication channel established by a socket can be terminated by using the `close()` system call, except that if the socket is stream-based, the closure on the socket will block until all the data has been transmitted. Alternatively, the shutdown() function can be used.

Function prototype for the `shutdown()` function:

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

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

The first parameter `sockfd` refers to the socket to be shut down and the second parameter `how` indicates the way the socket should be shut down. If `how` is set to 0, further reads will be disallowed; if `how` is 1, further writes will be disallowed; if `how` is 2, further reads and writes will be disallowed.

3 Assessed Laboratory Tasks

3.1 Task 1 (50%)

In this task, we will re-implement the client-server program from Section 2.2.2 using a message queue. Download the code for the `msq_server.c` and `msq_client.c` programs.

Complete the partial code given for the server program `msq_server.c`, and the client program `msq_client.c`, so that the following are achieved:

- The server program creates a message queue in order to receive messages sent from the client program.
- The client program reads data from standard input and sends the data as messages of "Type 1" to the server.
- Messages of "Type 1" received by the server program will be printed to standard output.
- A message of "Type 2" is used by the client to inform the server that there are no more messages. You might find the `Ctrl+D` key combination useful.

Note: Similar to what you have done in Section 2.2.2, run the server program first before running the client program.

3.2 Task 2 (50%)

In this task, we will re-implement the client-server program from Section 2.2.2, but by using sockets instead. Download the code for the `socket_server.c` and `socket_client.c` programs.

Complete the partial code given for the server program `socket_server.c`, and the client program `socket_client.c`, so that the following are achieved:

- The server program creates a socket in order to receive data sent from the client program.
- The client program creates a socket in order to send data to the server program.
- The server program reads from the socket and prints all data received to standard output. When an EOF character is read, the server should close its end of the socket.
- The client program reads data from standard input and sends the data to the socket. When an EOF character is read from standard input, the client should close its end of the socket. Again, you might find the `Ctrl+D` key combination useful.

Note: Once again, run the server program first before running the client program.