**INTRODUCTION TO INTER PROCESS COMMUNICATION (IPC)**

* Pipes
* FIFOs

Interprocess Communication

Diagram

Description automatically generated

An overview of communication options:

1. *Shared* *memory* permits processes to communicate by simply reading and writing to a shared memory page.
2. *Mapped* *memory* is similar to shared memory, except that it is associated with a file in the filesystem.
3. *Pipes* permit sequential communication from one process to a related process.
   1. Threads in same process or process with a child
4. *FIFOs* are similar to pipes, except that unrelated processes can communicate because the pipe is given a name in the filesystem.
5. *Sockets* support communication between unrelated processes even on different computers.

and more (message queues, etc)…

IPC criteria:

* Whether they restrict communication to related processes (processes with a common ancestor), to unrelated processes sharing the same file system, or to any computer connected to a network
* Whether a communicating process is limited to only writing data or only reading data
* The number of processes permitted to communicate
* Whether the communicating processes are synchronized by the IPC; for example, a reading process halts until data is available to read

An overview of synchronization options:

1. *Semaphore*: a kernel maintained non-negative integer that can solve many of the commonly encountered synchronization issues.
   1. *File locks*: are a synchronization method explicitly designed to coordinate the actions of multiple processes operating on the same file.
2. *Mutexes & condition variables (i.e. monitors)*: are intended to be used between threads and can solve (in theory) any synchronization issue.

More on these after the midterm exam/project.

IPC continues to grow beyond POSIX and UNIX and has led to the development of modern technologies such as:

* Remote procedure calls (Java RMI, JSON-RPC, SOAP, .net remote)
* Distributed object models (CORBA, sessions)

and many more.

*Pipes* are the oldest Unix IPC method.

A pipe is a communication device that permits unidirectional communication. Data written to the “write end” of the pipe is read back from the “read end.”

Pipes are serial devices; the data is always read from the pipe in the same order it was written. All of them will be queued inside the pipe until memory allocated for that pipe is fulled.

Typically, a pipe is used to communicate between two threads in a single process or between parent and child processes.

In a shell, the symbol | creates a pipe. For example, this shell command causes the shell to produce two child processes, one for ls and one for wc (word count):



This sends std output of whatever comes from ls to the wc command and display it.

The shell also creates a pipe connecting the standard output of the ls subprocess with the standard input of the wc process. The filenames listed by ls are sent to wc in exactly the same order as if they were sent directly to the terminal.

A picture containing diagram

Description automatically generated

The following commond line inputs have the same effect:

Diagram

Description automatically generated

A pipe is simply a buffer maintained in kernel memory. This buffer has a finite and limited capacity (often 64KB).

If the writer process writes faster than the reader process consumes the data, and if the pipe cannot store more data, the writer process blocks until more capacity becomes available.

If the reader tries to read but no data is available, it blocks until data becomes available. Thus, the pipe automatically synchronizes the two processes.

* We do this one most of the time bc it not easy to ensure pipe is full of data so writer will be blocked.
* Waiting for empty pipe can easily be done.

The pipe() system call creates a new pipe.

Diagram

Description automatically generated

Returns 0 on success, or –1 on error.

First supply an integer array of size 2. Then the call to pipe() stores the reading file descriptor in array position 0 and the writing file descriptor in position 1.

If you want to form a pipe between child and parent processes, after the fork, parent and child both have the same pipe. One of them has to remove write side and one of them has to remove read side in order to communication between them will be done properly.

Example:

Text

Description automatically generated

Data written to the file descriptor read\_fd can be read back from write\_fd.

A process’s file descriptors cannot be passed to unrelated processes; however, when the process calls fork, file descriptors are copied to the new child process. Thus, pipes can connect only related processes.

Diagram

Description automatically generated

If child wants to send data back to parent, it can form another pipe or some other IPC structure.

(a) can also be done but it is hard to manipulate whats going on.

Table

Description automatically generated with medium confidence

Graphical user interface, text, application

Description automatically generated

Graphical user interface, text, application, email

Description automatically generated

When a pipe is created, the file descriptors used for the two ends of the pipe are the next lowest-numbered descriptors available. Since, in normal circumstances, descriptors 0, 1, and 2 are already in use for a process, some higher-numbered descriptors will be allocated for the pipe.

So how do we bring about the situation where two programs that read from stdin and write to stdout are connected using a pipe, such that the standard output of one program is directed into the pipe and the standard input of the other is taken from the pipe?

By duplicating file descriptors!

duplicate stdin, remove original stdin, use duplicated one

Text

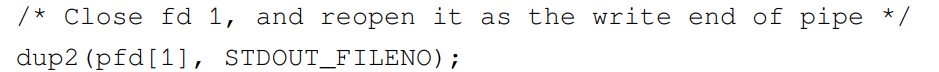
Description automatically generated

The end result of the above steps is that the process’s standard output is bound to the write end of the pipe. A corresponding set of calls can be used to bind a process’s standard input to the read end of the pipe.

However dup assumes that 0, 1 and 2 are already open.

If however 0 was closed for some reason, pfd[1] would become its clone instead of 1.

That’s why it’s a better idea to use dup2() instead of close() and dup()



After duplicating pfd[1], we now have two file descriptors referring to the write end of the pipe: descriptor 1 and pfd[1]. Since unused pipe file descriptors should be closed, after the dup2() call, we close the superfluous descriptor:

Text

Description automatically generated with medium confidence

Graphical user interface, text, application

Description automatically generatedText, application

Description automatically generated

parent removes pipe from itself and waits for its children

ls | wc -l 🡪 mimicking this example. First child is using ls command and closing read part. Second child is closing write part. We redirect 2nd child’s stdin and execute wc -l command.

The inherent synchronization mechanism of pipes can be exploited for general purpose synchronization as well.

Imagine a parent process that wants to wait/block until all of its children have accomplished their respective tasks (a.k.a. synchronization barrier); could you solve this through signals?

It’s solvable through pipes as well: the parent builds a pipe before creating the child processes.

Each child inherits a file descriptor for the write end of the pipe and closes this descriptor once it has completed its action.

After all of the children have closed their file descriptors for the write end of the pipe, the parent’s read() from the pipe will complete, returning end-of-file (0). At this point, the parent is free to carry on to do other work.

(Note that closing the unused write end of the pipe in the parent is essential for the correct operation of this technique; otherwise, the parent would block forever when trying to read from the pipe.)

Text

Description automatically generated

Text

Description automatically generated

Parent will do the reading so child is closing read part.

closing write part means that no one is writing to pipe now.

Text, letter

Description automatically generated

After here, we know no other processes doing anything. All other processes dead.

parent will read until EOF

A common use for pipes is to execute a shell command and either read its output or send it some input. The popen() and pclose() functions are provided to simplify this task.

Graphical user interface, text, application

Description automatically generated

The popen() function creates a pipe, and then forks a child process that execs a shell, which in turn creates a child process to execute the string given in command.

The mode argument is a string that determines whether the calling process will read from the pipe (mode is r) or write to it (mode is w).

Diagram

Description automatically generated

Text

Description automatically generated

Security-wise, popen() is risky (like system()) as it involves executing a shell (creates a pipe on the shell and executes commands on the shell, closes it and come back); you can find various exploit examples online for both, usually involving executing commands with a high-privilege user account.

popen may not work on every system. Systems may have different versions of shell.

You may also open high-privileged shell which can be used by other processes that have lower privileges.

Our second IPC tool is **FIFO**.

A FIFO is simply a pipe that has a name in the filesystem.

Any process can open or close the FIFO; the processes on either end of the pipe need not be related to each other. FIFOs are also called named pipes.

Just as with pipes, a FIFO has a write end and a read end, and data is read from the pipe in the same order as it is written. This fact gives FIFOs their name: first-in, first-out.

We can create a FIFO from the shell using the mkfifo command. Specify the path to the FIFO on the command line. For example, create a FIFO in /tmp/fifo by invoking this:

A picture containing text

Description automatically generated

When applied to a FIFO (or pipe), fstat() and stat() return a file type of S\_IFIFO in the st\_mode field of the stat structure. When listed with ls –l, a FIFO is shown with the type p in the first column.

Any other process can reach the file and see contents of it if they have the privileges.

In one window, read from the FIFO by invoking the following:



In a second window, write to the FIFO by invoking this:



Then type in some lines of text. Each time you press Enter, the line of text is sent through the FIFO and appears in the first window. Close the FIFO by pressing Ctrl+D in the second window.

When you read from it, read stuff will be gone bc you use a pipe.

Remove the FIFO with this line:



The mkfifo() call creates a new FIFO with the given pathname.

Text

Description automatically generated with medium confidence

The mode argument specifies the permissions for the new FIFO. These permissions are specified by OR’ing the desired combination of constants: S\_I(R|W|X)(USR|GRP|OTH)

Access a FIFO just like an ordinary file. To communicate through a FIFO, one process must open it for writing, and another process must open it for reading. Either low-level I/O functions (open, write, read, close) or C library I/O functions (fopen, fprintf, fscanf, fclose) may be used.

For example, to write a buffer of data to a FIFO using low-level I/O routines, you could use this code:

Text

Description automatically generated

To read a string from the FIFO using C library I/O functions, you could use this code:

Text

Description automatically generated

A FIFO can have multiple readers or multiple writers. Bytes from each writer are written atomically up to a maximum size of PIPE\_BUF (4KB on Linux). Chunks from simultaneous writers can be interleaved. Similar rules apply to simultaneous reads.

The FIFO must be opened on both ends before data can be passed. Normally, opening the FIFO blocks until the other end is opened also (use the O\_NONBLOCK flag in open() if you don’t want this).

Graphical user interface, text, application

Description automatically generated

Graphical user interface, text, application

Description automatically generated

We are not closing the file descriptors this time. You gotta be synchronous between reading and writing. You cannot have multiple files for reading, multiple for writing to FIFO. 1 for each.

Graphical user interface, text, application

Description automatically generated

Pipes, fifos and the client-server model

The client–server model is a distributed application structure that partitions tasks between the providers of a resource or service, called servers, and service requesters, called clients.

e.g. http server and web browser

This is opposed to the peer-to-peer (p2p) model, where two or more programs/computers (or simply peers) pool their resources and communicate in a decentralized system (each peer can act as both client and server).

e.g. bittorrent based file sharing

Client-server models have many flavors depending on the number of servers, number of clients, type of communication between them (simple request, request-reply), way of handling the clients (iterative, concurrent), etc.

Consider the common scenario of a single server with multiple clients using fifo based communication, where the server provides the (trivial) service of assigning unique sequential numbers to each client that requests them.

The first impulse is usually to create a fifo between the server and the clients...this is bad idea; why?

Because multiple clients would race to read from the FIFO, and possibly read each other’s response messages rather than their own.

Therefore, each client creates a unique FIFO that the server uses for delivering the response for that client, and the server needs to know how to find each client’s FIFO; either with a pre-agreed name template or by sending the fifo name as part of the request message.

Diagram

Description automatically generated

For instance the names of the client fifos can consist of the pids of the clients, thus easily ensuring uniqueness.

Client forms the client FIFO.

Another practical issue to consider that the data in pipes and FIFOs is a byte stream; boundaries between multiple messages are not preserved. This means that when multiple messages are being delivered to a single process, such as the server in our example, then the sender and receiver must agree on some convention for separating the messages; e.g.

Diagram

Description automatically generated

Text

Description automatically generated

Text

Description automatically generated

Text

Description automatically generated

Text, letter

Description automatically generated

Text

Description automatically generated