Pipes

IPC

> *Interprocess Communication(IPC) is the generic term describing how two processes may exchange information with each other.*
> *This communication may be an exchange of data for which two or more processes are cooperatively processing the data or synchronization information to help two independent, but related, processes schedule work so that they do not destructively overlap.*

Examples of IPC are pipes and signals.

How is IPC achieved?

IPC occcurs through some shared resource. In the case of pipe between two proc on same system, shared resource is OS. For two systems, its the network. Both procs need to have access to IPC facilities. For example, When two or more procs reside on the same host they share:

- memory
- file space
- communication facilities
- signals

Pipes

> *Pipes are an interprocess communication mechanism that allow two or more processes to send information to each other.*

In `who | wc -l`, the output of `who` is *piped* into `wc`.

Pipes have a reader and a writer. Both can use the pipe concurrently. Pipe buffers o/p of writer and suspends it if its full. The OS regulates the flow of data through the pipe. If pipe is empty, it suspends the reader. We have *named* and *unamed* pipes.

Unnamed pipes: `pipe()`

> *An unnamed pipe is a unidirectional communications link that automatically buffers its input(the maximum size of the input varies with different versions of UNIX, but is approximately 5K) and may be created using the "pipe()" system call.*

Both ends of the pipe are file descriptors. So they are read from/written to using `read()` and `write()` system calls. And it is closed using the `close()` system call.

A process can pipe to itself (why? `¯\_(ツ)_/¯`). As pipes are f-ds they are shared between child and parent procs. This can lead to a mess of pipes between children and parents. This is fixed by closing of the ends of the pipe that are not required. If you want bi-directional communinication, open up another pipe.

System Call: `int pipe(int fd[2])`

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

- That code creates a pipe. `fd[0]` reads to, `fd[1]` writes from the pipe.
- If a process whose write end is closed, tries to read, nothing happens.
- If a process reads from an empty pipe, it is suspended till some input is there.
- If a process tries to read more than there is there in pipe, all the data is outputed.
- If a process write to a pipe whose read end is closed, the write fails and the writer is sent a `SIGPIPE` signal(default action is to terminate the writer).
- If a process writes fewer bytes to a pipe than the pipe can hold, the the write is gauranteed to occur withought any other process preempting it i.e the write will be atomic.
- If OS can't make a new pipe, `pipe()` returns a -1.

Typical sequence of events for communication b/w parent and child.

1. The parent process creates an unnamed pipe using `pipe()`
2. The parent forks.
3. The writer/reader closes their end.
4. The processes communicates by using `write()` and `read()`
5. Each process closes its active pipe descriptor when it's finished with it

B-directional communication is only possible with *two* pipes.

Example program:

```
#include <stdio.h>
#define READ 0 /* The index of the “read” end of the pipe */
#define WRITE 1 /* The index of the “write” end of the pipe */
char* phrase =“Good morning”;

int main() {
    int fd[2], bytesRead;
    char message[100]; /* Parent process’ message buffer */
    pipe(fd); /* Create an unnamed pipe */
    if ( fork() == 0 ) /* Child, write */
    {
        close(fd[READ]); /* Close unused end */
        write(fd[WRITE], phrase, strlen(phrase)+1); /* Send */
        close(fd[WRITE]); /* Close used end */
    } else /* Parent, reader */ {
        close(fd[WRITE]); /* Close unused end */
        bytesRead = read( fd[READ], message, 100 ); /* Receive */
        printf(“Read %d bytes: %s \n”, bytesRead, message );
        close(fd[READ]); /* Close used end */
    }
}
```

When a writer process sends more than one variable length message into the pipe, it must use some sort of protocol to tell the reader when the message ends. For example, sending the length of the data before the data itself or ending the message with a predecided delimiter.

Named pipes

> *Named pipes, often referred to as FIFOs(first in, first out), are less restricted than unnamed pipes.*

They offer the following advantages:

- They have a name that exists in the file system.
- They may be used by unrelated processes.
- They exist until explicitly deleted.

Its created by using the UNIX `mknod` utility or `mknod()` system call.

`mknod` utility

Use the `p` option to creat a pipe, Set the permissions of the pipe using `chmod`.

```
mknod myPipe p     # create pipe
chmod ug+rw myPipe # update permissions
ls -lg myPipe      # examine attributes
# prw-rw---- 1 ...
```

`mknod()` system call

Use the `S_IFIFO` option to use `mknod()` to make a named pipe.

```
mknod("myPipe", S_IFIFO, 0);
chmod("myPipe", 0660);
```

Read/write from the pipe using `read()`/`write()`. Close it using `close()` and remove it from the file system using `unlink()`. Names pipes are also unidirectional. Writer/reader procs should only use it for writing/reading. A reader/writer will have to wait until some other proc has opened the other end of the pipe for writing/reading. Named pipes don't work across nets.
