Pipes

- http://tldp.org/LDP/lpg/node9.html
- http://beej.us/guide/bgipc/output/html/multipage/pipes.html
-

Pipes

- M. D. McIlroy - October 11, 1964
  - We should have some ways of coupling programs like garden hose--screw in another segment when it becomes when it becomes necessary to massage data in another way This is the way of IO also.
- In Unix pipes are the original inter-process communication mechanisms

Redirect in the shell

- ls >foo
  - sends the output of the directory lister **ls** to a file named ‘foo’.
- wc < foo
  - causes the word-count utility wc(1) to take its standard input from the file ‘foo’,
  - and deliver a character/word/line count to standard output.

Pipes in the shell

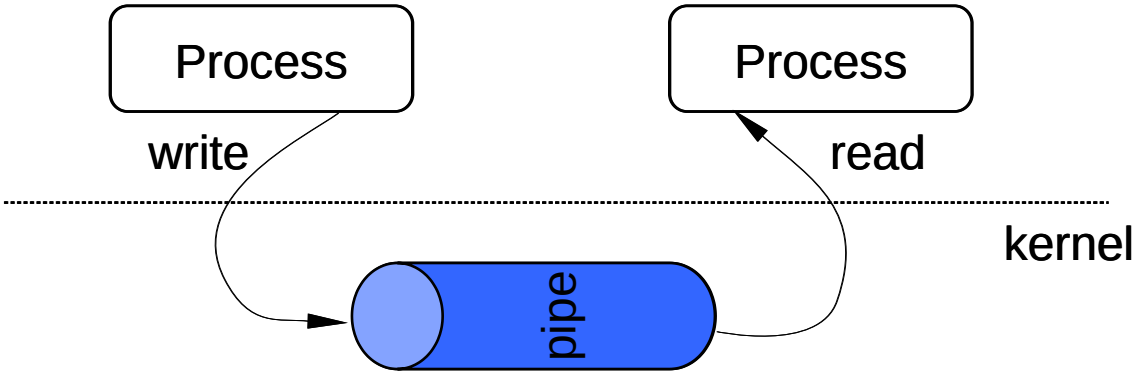
- pipe operation
  - connects the standard output of one program to the standard input of another.
  - A chain of programs connected in this way is called a pipeline.
- ls | wc
  - Counts character/word/line count for the current directory listing
- tr -c '[:alnum:]' '[\n*]' | sort -iu | grep -v '^[0-9]*$'

Pipes in the shell

- All the stages in a pipeline run concurrently.
  - Each stage waits for input on the output of the previous one,
  - no stage has to exit before the next can run.
- It is unidirectional.
  - p1 | p2 p1 → p2 p1 ← p2
  - Impossible to pass information back
    - Just p2 dead notification
- Protocol for passing data
  - is simply the receiver's input format.

Pipes

- Read/Writes
  - File operations
- Processes
  - Should be related
    - Father/soon
    - Brothers



Pipes

- Pipe creation:
  - int pipe(int fd[2]);
  - int pipe2(int pipefd[2], int flags); /* O_NONBLOCK */
- Opens two files
  - fd[0] descriptor open for reading
  - fd[1] descriptor open for writing
- Returns
  - 0 successful
  - 1 unsuccessful (erno variable set)
- Pipes can only connect processes with a common ancestor
- Pipe information is managed as a open file

Pipes

- Communication (data read/write)
  - ssize_t read(int fd, void *buf, size_t count);
  - ssize_t write(int fd, void *buf, size_t count);
- 1st argument
  - File descriptor (fd[0] or fd[1])
- 2nd argument data buffer address (data destination/source)
- 3rd argument number of bytes to read/write
- Return number of bytes read/written
- Blocks or not (O_NONBLOCK)

Pipes

- If a process attempts to read from an empty pipe,
  - then read(2) will block until data is available.
- If a process attempts to write to a full pipe,
  - then write(2) blocks until sufficient data has been read from the pipe to allow the write to complete.
- Nonblocking I/O is possible
  - using O_NONBLOCK status flag.
- The communication channel provided by a pipe is a byte stream:
  - there is no concept of message boundaries.
-

fd vs FILE *

fd	FILE *
Unix system calls	C library
man 2 intro	man 3 intro
• open	• fopen
• read/write	• fread/fwrite
Byte stream	Mostly character streams
unbuffered	buffered
atomic	??????
fileno	fdopen
#include <sys/types.h> #include <sys/stat.h> #include <fcntl.h>	#include <stdio.h>

Pipes

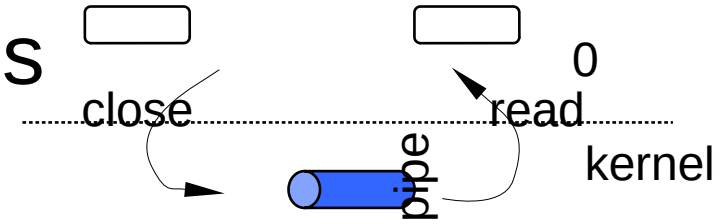
- Messages are limited to byte streams.
- Information flow is unidirectional
  - One process reads one process writes
  - Uses file descriptors functionality
- Major limitation
  - Processes should be related
- How to implement pipes that are accessible by other processes?
  - Giving them a name
  - Registering them in the File system

Pipes

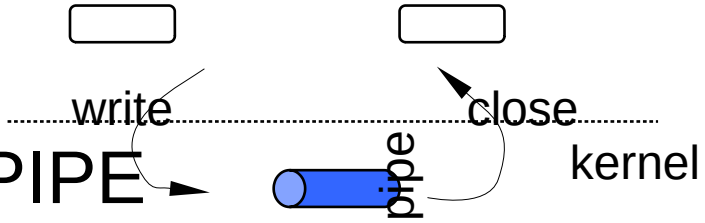
- A pipe has a limited capacity.
  - If the pipe is full, then a write(2) will block or fail, depending on whether the O_NONBLOCK flag is set
- Different implementations have different limits for the pipe capacity.
  - Applications should not rely on a particular capacity
  - application should consume data as soon as possible
- POSIX.1-2001 says that write(2)s of less than PIPE_BUF bytes must be atomic:
  - the output data is written to the pipe as a contiguous sequence.
  - Writes of more than PIPE_BUF bytes may be nonatomic:
    - the kernel may interleave the data with data written by other processes.
  - POSIX.1-2001 requires PIPE_BUF to be at least 512 bytes.
    - On Linux PIPE_BUF is 4096 bytes.

Closing Pipes

- Closing all write ends
  - Read will return 0



- Closing all read ends
  - Write will produce SIGPIPE



- After fork processes should close not needed ends
  - For previous notifications to work

Pipes

- Implementation – Kernel / syscall
- Scope - local
- No Duplex
- Time-coupling
- Space-coupling +-
- Explicit
- Synchronization – Yes by default
- Process relation - related
- Identification - NA
- API – file operations

FIFO / Named Pipes

- http://tldp.org/LDP/lpg/node15.html
- http://beej.us/guide/bgipc/output/html/multipage/fifos.html
-

FIFO / Named Pipes

- To solve Pipes limitations
  - FIFOs were defined
    - Also referred as named pipes
- Can be used by unrelated processes
- Are referred and identified by a file in the file system
- A FIFO is special file similar to a pipe,
  - That is created in a different way
  - Instead of being an anonymous communications channel,
    - FIFO is entered into the file system by calling mkfifo()
  -

FIFO / Named Pipes

- FIFO creations
  - int mkfifo(const char *pathname, mode_t mode);
- 1st argument
  - FIFO name (full path)
- 2nd argument
  - Access permissions (like a regular file)
- Once you have created a FIFO special file in this way, any process can open it for reading or writing,
  - in the same way as an ordinary file.
- **3151348 0 prw-r--r-- 1 jnos users 0 Mar 22 10:05 test_fifo**
- On success mkfifo() returns 0.
  - In the case of an error, -1 is returned (errno is set appropriately).

FIFO / Named Pipes

- Before being used the FIFO should be opened
  - int open(const char *pathname, int flags);
- 1st argument
  - FIFO name
- 2nd argument
  - Bits that define access mode
  - O_RDONLY (just reading)
  - O_RDONLY (just writing)
  - O_NONBLOCK (non blocking I/O)
- The return value is
  - -1 in case of error
  - Or a positive file descriptor

FIFO / Named Pipes

- A FIFO has to be opened at both ends simultaneously before you can proceed to do any input or output operations on it.
  - Opening a FIFO for reading normally blocks until some other process opens the same FIFO for writing, and vice versa.
- Opening the FIFO in O_NONBLOCK mode
  - Returns success if other process has already opened
  - Returns -1 if it is the first open
    - Sets errno to ENXIO

FIFO / Named Pipes

- Communication (data read/write)
  - ssize_t read(int fd, void *buf, size_t count);
  - ssize_t write(int fd, void *buf, size_t count);
- 1st argument
  - File descriptor (fd[0] or fd[1])
- 2nd argument data buffer address (data destination/source)
- 3rd argument number of bytes to read/write
- Return number of bytes read/written
- Blocks or not (O_NONBLOCK)

FIFO / Named Pipes

- If a process attempts to read from an empty FIFO,
  - then read(2) will block until data is available.
- If a process attempts to write to a full FIFO,
  - then write(2) blocks until sufficient data has been read from the pipe to allow the write to complete.
- Nonblocking I/O is possible
  - using O_NONBLOCK status flag.
- The communication channel provided by a FIFO is a byte stream:
  - there is no concept of message boundaries.
- The communication is unidirectional

Closing FIFOS

- Closing all write ends
  - Read will return 0
- Closing all read ends
  - Write will produce SIGPIPE
- After fork processes should close not needed ends
  - For previous notifications to work

FIFO / Named Pipes

- Implementation – Kernel / syscall
- Scope - local
- No Duplex
- Time-coupling
- Space-coupling +-
- Explicit
- Synchronization – Yes by default
- Process relation - unrelated
- Identification – file name
- API – file operations