Introduction to Pipes

- **History of Pipes:** Pipes history goes back to 3rd edition of UNIX in 1973. They have no name and can therefore be used only between related processes. This was corrected in 1982 with the addition of FIFOs
- **Byte stream:** When we say that a pipe is a byte stream, we mean that there is no concept of message boundaries when using a pipe. Each read operation may read an arbitrary number of bytes regardless of the size of bytes written by the writer. Furthermore, the data passes through the pipe sequentially, bytes are read from a pipe in exactly the order they were written. It is not possible to randomly access the data in a pipe using `lseek()`
- **Pipes are unidirectional:** Data can travel only in one direction. One end of the pipe is used for writing, and the other end is used for reading

Reading from a pipe:

- When a process reads bytes from a pipe, those bytes are removed from the pipe (Destructive read semantics)
- By default, when a process attempts to read from a pipe that is currently empty, the read call blocks until some bytes are written into the pipe
- If the write end of a pipe is closed, and a process tries to read, it will receive and EOF character, i.e., `read()` returns 0
- If two processes try to read from the same pipe, one process will get some of the bytes from the pipe, and the other process will get the other bytes. Unless the two processes use some method to coordinate their access to the pipe, the data they read are likely to be incomplete

Writing to a pipe:

- By default, when a process attempts to write to a pipe that is currently full, the `write(2)` call blocks until there is enough space in the pipe
- If a process tries to write, say, 1000 bytes, and there is room for 500 bytes only, the call waits until 1000 bytes of space are available
- If the read end of a pipe is closed and a process tries to write, the kernel sends `SIGPIPE` to the writer process

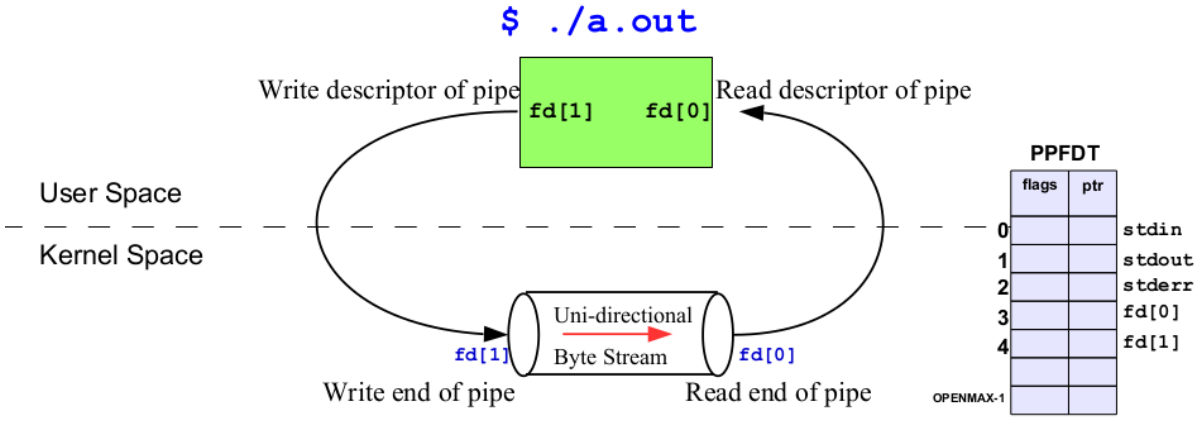
- **Size of Pipe:**
  - If multiple processes are writing to a single pipe, then it is guaranteed that their data won't be intermingled if they write no more than `PIPE_BUF` bytes at a time
  - This is because writing `PIPE_BUF` number of bytes to a pipe is an atomic operation. On Linux, value of `PIPE_BUF` is 4096
  - When writing more bytes than `PIPE_BUF` to a pipe, the kernel may transfer the data in multiple smaller pieces, appending further data as the reader removes bytes from the pipe. The `write()` call blocks until all of the data has been written to the pipe
  - When there is a single writer process, this doesn't matter. But in case of multiple writer processes, this may cause problems

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

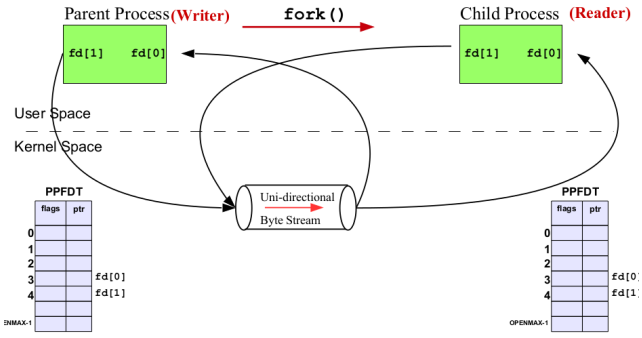
- A pipe is created by calling the `pipe()` system call
- Creating a pipe is similar to opening two files. A successful call to `pipe()` returns two open file descriptors in the array `fd`; one contains the read descriptor of the pipe, `fd[0]`, and the other contains the write descriptor of the pipe `fd[1]`
- As with any file descriptor, we can use the `read()` and `write()` system calls to perform I/O on the pipe. Once written to the write end of a pipe, data is immediately available to be read from the read end. A `read()` from a pipe blocks if the pipe is empty
- From an implementation point of view, a pipe is a fixed-size main memory circular buffer created and maintained by the kernel. The kernel handles the synchronization required for making the reader process wait when the pipe is empty and the writer process wait when the pipe is full

Use of Pipe in a Single Process

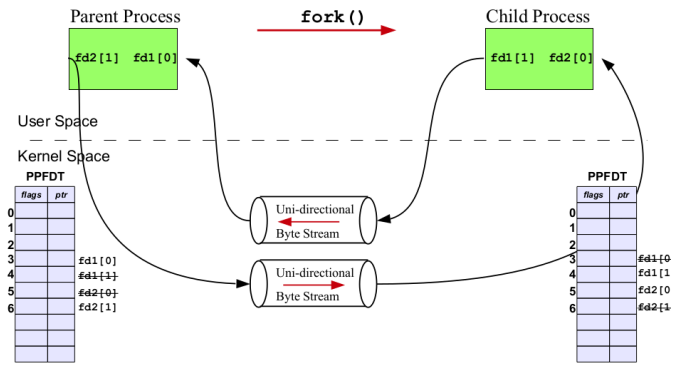
```
int fd[2];
pipe(fd);
int cw = write(fd[1],msg, strlen(msg));
int cr = read(fd[0], buf, cw);
write(1, buf, cr);
```



Use of Pipe Between two Related Processes



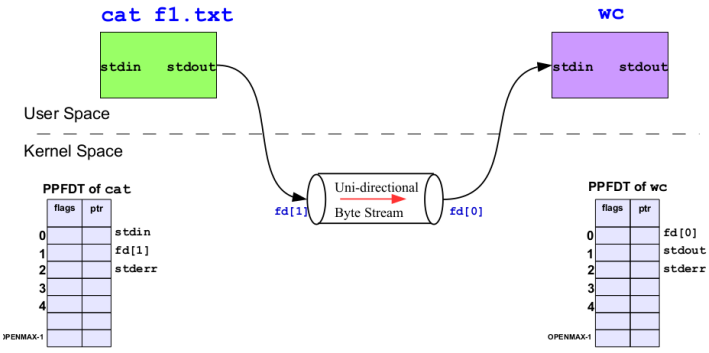
Bidirectional Comm Using Pipes



Example: `cat f1.txt | wc`

Let us try writing a program that simulate the shell command

```
cat f1.txt | wc
```



Example: `man ls | grep ls | wc -l`