Interprocess Communications Using Pipes

- UNIX provides a construct called the **pipe,** which is a one-way communications channel which couples one process to another, and is yet another generalization of the UNIX file concept.
- A process can send data 'down' the pipe by using the write system call, and another process can receive the data by using read at the other end.
- Pipes are one of the strongest and most distinctive features of UNIX, especially at command level. They allow arbitrary sequences of commands to be simply coupled together.
- For example,

  $ who | wc -l

  pipes the output of the who into the word count program ***wc***, the ***-l*** option telling **wc** just to count lines. The number finally output by ***wc*** is a count of the number of logged-on users.

Programming with Pipes

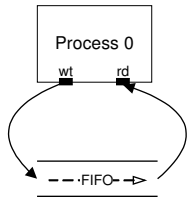
- Within a program a pipe is created using a system call named pipe. If successful, this call returns two file descriptors: one for writing down the pipe, and one for reading from it.

  Usage

  int filedes[2], retval;

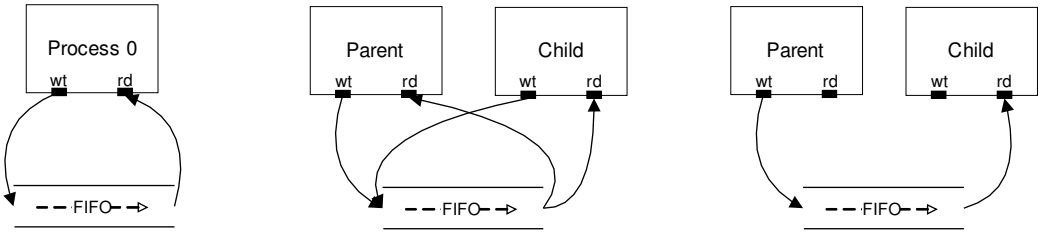
  retval = pipe(filedes);

- **filedes** is a two-integer array that will hold the file descriptors that will identify the pipe.
- If the call is successful, **filedes[0]** will be open for **reading** from the pipe and **filedes[1]** will be open for **writing** down it.
- pipe can fail and so return -1. This can happen if the call would cause more file descriptors to be opened than the per-process limit or if the kernel's open file table would overflow.
- Once created, a pipe can be straightforwardly manipulated with read and write. The first example (pipe_single_process.c) shows this; it creates a pipe, writes three messages down it, then reads them back.



pipe with single process

- Pipes treat data on a first-in first-out or FIFO basis. In other words, what you place first into a pipe is what is read first at the other end.
- A pipe's true value only becomes apparent when it is used in conjunction with the fork system call, where the fact that file descriptors remain open across a fork can be exploited.
- The next example (pipe_with_fork.c) creates a pipe and calls fork. The parent process then writes a series of messages down to the child.



1. pipe with single process **2. pipe after fork ()** **3. after closing unused descriptors**

The size of a pipe

- In practice, it is important to note that the size of a pipe is finite. The limit is 5120 bytes.
- If write is made on a pipe and there is enough space, then the data is sent down the pipe and the call returns immediately.
- If however a write is made that would overfill the pipe, process execution is (normally) suspended until room is made by another process reading from the pipe.
- When a read call is made, the system checks whether the pipe is empty. If it is empty, the read will (normally) block until data is written into the pipe by another process.
- If there is data waiting in the pipe, then the read returns, even if there is less data than the amount requested.

Closing down pipes

1. **Closing the write-only descriptor.**
   - If there are other processes that still have the pipe open for writing, then the system will take no other action on the pipe.
   - If there are no more processes capable of writing to the pipe, the write descriptor will be closed. Processes that were asleep waiting to read from the pipe will be woken up, and their read calls returns zero. The effect for the reading processes is therefore much like reaching the end of an ordinary file.
2. **Closing the read-only file descriptor.**
   - If there are still processes that have the pipe open for reading, then again the system will take no other action on the pipe.
   - If no other process is reading the pipe, however, the read descriptor will be closed and all other processes waiting to write to pipe are sent the signal **SIGPIPE** by the kernel. Processes that attempt to write to the pipe later will also be sent **SIGPIPE**.

Non-blocking Reads and Writes

- As we have seen both read and write can block when used on a pipe. Sometimes this isn't desirable.
- You may want a program, for example, to execute an error routine, or maybe poll through several pipes until it receives data through one of them. There are a couple of ways of ensuring that a read or write on a pipe won't hang.

- The first method is to use ***fstat*** on the pipe. The **st_size** field in the returned structure gives the number of characters currently in the pipe.
- If only one process is reading the pipe, this is fine. However if several processes are reading the pipe, another process could read from a pipe in the gap between ***fstat*** and read.
- The second method is to use ***fcntl***. Among its many roles, it allows a process to set the **O_NDELAY** flag for a file descriptor.
- This stops a future read or write on a pipe from blocking. In this context, ***fcntl*** can be used as follows:

```
#include <fcntl.h>
.
.
if ( fcntl ( filedes, F_SETFL, O_NDELAY ) < 0 )
        perror ( "fcntl" );
```

- If **filedes** was the write-only file descriptor for a pipe, then future calls to write would never block if the pipe was full. They would instead return a value of 0 immediately.
- Similarly, if **filedes** represented the reading end of a pipe, then the process would immediately return a value of 0 if there was no data in the pipe, rather than sleeping.
- The example program (pipe_nonblocking.c) demonstrates this variation on the ***fcntl*** theme. It creates a pipe, sets the **O_NDELAY** flag for the read file descriptor, then forks.
- The child process sends messages to the parent, which sits in a loop polling the pipe to see if any data has arrived.
- Notice the use of special message to terminate the conversation. A process has no way to distinguish between a pipe that is empty, and one where the write end has been closed.
- The output of this example isn't exactly predictable since the number of 'pipe empty' messages may vary. On one machine it produced:

```
(pipe empty)
MSG=hello
(pipe empty)
(pipe empty)
(pipe empty)
MSG=hello
(pipe empty)
(pipe empty)
(pipe empty)
MSG=hello
(pipe empty)
(pipe empty)
(pipe empty)
End of conversation
```