Network Programming

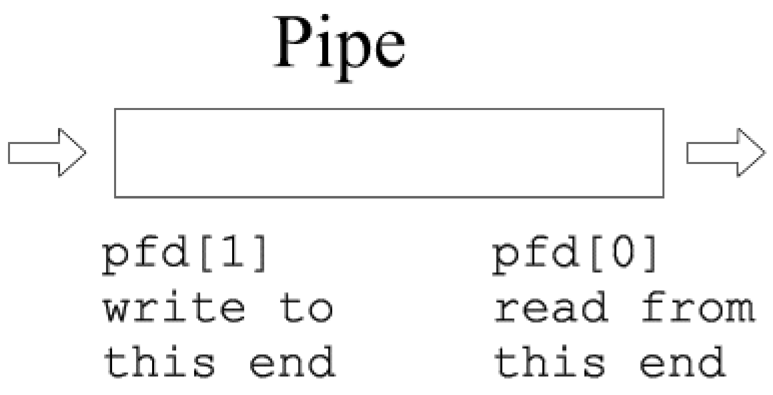
Unix Pipes

A pipe is used for one-way communication of a stream of bytes. The command to create a pipe is ***pipe()***, which takes an array of two integers. It fills in the array with two file descriptors that can be used for low-level I/O.

Creating a pipe

```
int pfd[2];

pipe(pfd);
```



I/O with a pipe

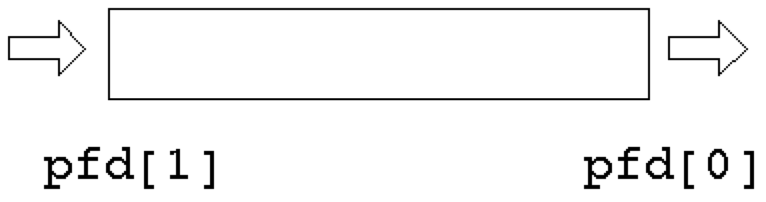
These two file descriptors can be used for block I/O

```
write(pfd[1], buf, size);
read(pfd[0], buf, SIZE);
```

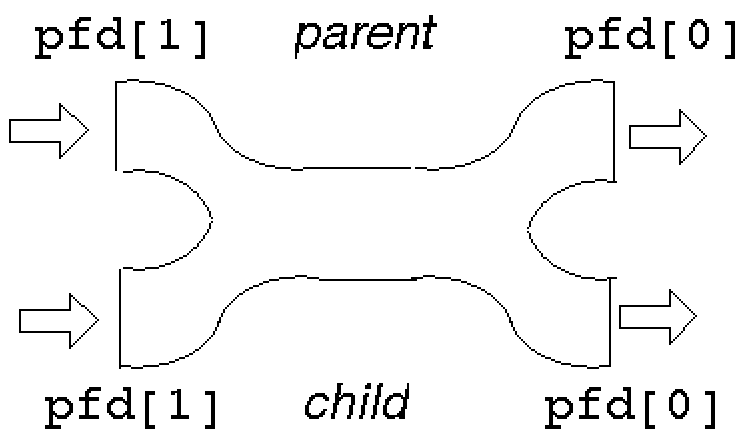
Fork and a pipe

A single process would not use a pipe. They are used when two processes wish to communicate in a one-way fashion. A process splits in two using ***fork()***. A pipe opened before the fork becomes shared between the two processes.

Before fork



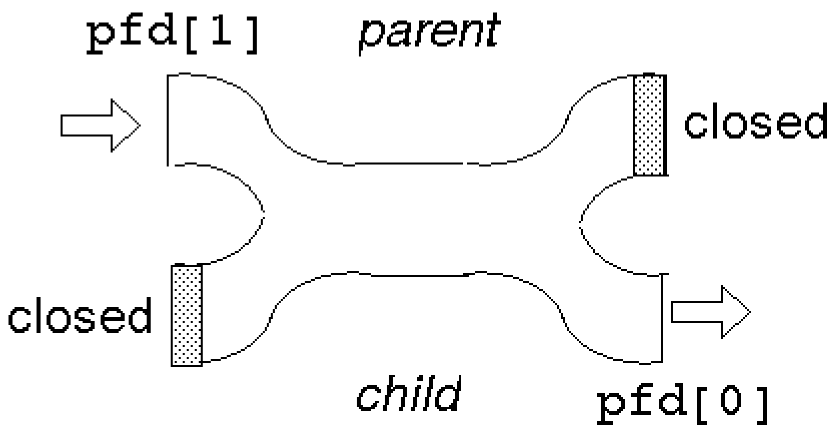
After fork



This gives *two* read ends and *two* write ends. The read end of the pipe will not be closed until both of the read ends are closed, and the write end will not be closed until both the write ends are closed.

Either process can write into the pipe, and either can read from it. Which process will get what is not known.

For predictable behaviour, one of the processes must close its read end, and the other must close its write end. Then it will become a simple pipeline again.



Suppose the parent wants to write down a pipeline to a child. The parent closes its read end, and writes into the other end. The child closes its write end and reads from the other end.

When the processes have ceased communication, the parent closes its write end. This means that the child gets eof on its next read, and it can close its read end.

```
#include <stdio.h>

#define SIZE 1024

int main(int argc, char **argv)
{
  int pfd[2];
  int nread;
  int pid;
  char buf[SIZE];

  if (pipe(pfd) == -1)
  {
    perror("pipe failed");
    exit(1);
  }
  if ((pid = fork()) < 0)
  {
    perror("fork failed");
    exit(2);
  }

  if (pid == 0)
  {
    /* child */
    close(pfd[1]);
    while ((nread =
         read(pfd[0], buf, SIZE))
         != 0)
      printf("child read %s\n", buf);
    close(pfd[0]);
  } else {
    /* parent */
    close(pfd[0]);
    strcpy(buf, "hello...");
    /* include null terminator in write */
    write(pfd[1], buf,
          strlen(buf)+1);
    close(pfd[1]);
  }
  exit(0);
}
```

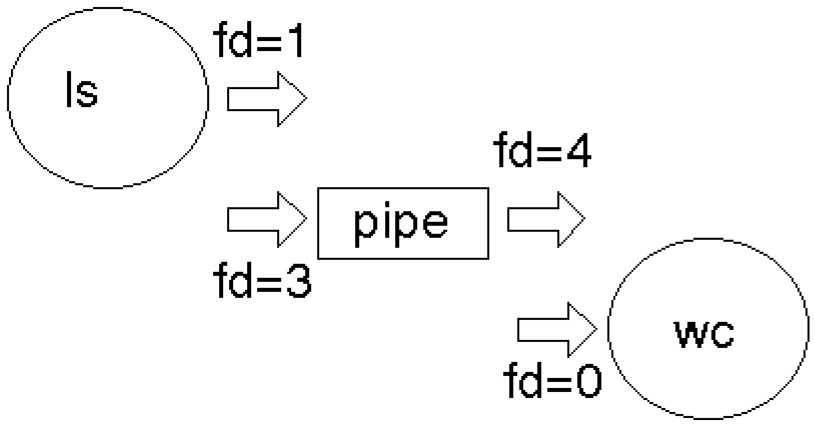
dup

A pipeline works because the two processes know the file descriptor of each end of the pipe. Each process has a stdin (0), a stdout (1) and a stderr (2). The file descriptors will depend on which other files have been opened, but could be 3 and 4, say.

Suppose one of the processes replaces itself by an ``exec''. The new process will have files for descriptors 0, 1, 2, 3 and 4 open. How will it know which are the ones belonging to the pipe? It can't.

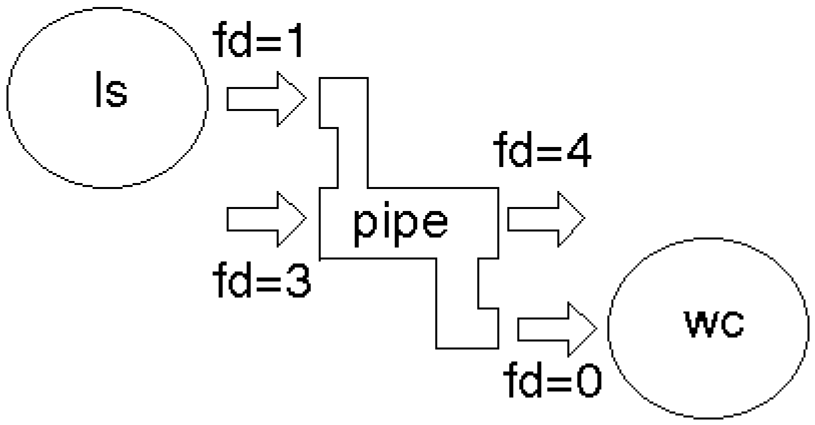
Example:

To implement ``ls | wc'' the shell will have created a pipe and then forked. The parent will exec to be replaced by ``ls'', and the child will exec to be replaced by ``wc'' The write end of the pipe may be descriptor 3 and the read end may be descriptor 4. ``ls'' normally writes to 1 and ``wc'' normally reads from 0. How do these get matched up?

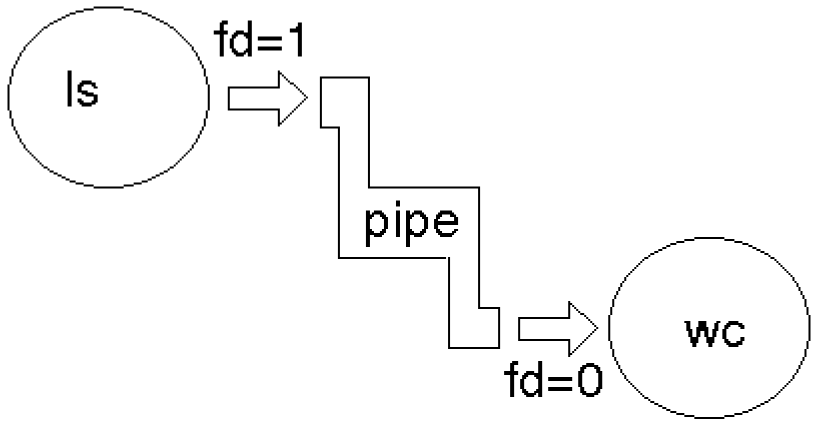


The ***dup2()*** function call takes an existing file descriptor, and another one that it ``would like to be". Here, fd=3 would also like to be 1, and fd=4 would like to be 0. So we dup2 fd=3 as 1, and dup2 fd=4 as 0. Then the old fd=3 and fd=4 can be closed as they are no longer needed.

After dup2



After close



Without any error checks, the program to do this is

```
int main(void)
{
  int pfd[2];

  pipe(pfd);
  if (fork() == 0) {
    close(pfd[1]);
    dup2(pfd[0], 0);
    close(pfd[0]);
    execlp("wc", "wc",
           (char *) 0);
  } else {
    close(pfd[0]);
    dup2(pfd[1], 1);
    close(pfd[1]);
    execlp("ls", "ls",
           (char *) 0);
  }
}
```

With checks, it is

```
#include <stdio.h>

int main(void)
{
  int pfd[2];
  int pid;

  if (pipe(pfd) == -1)
  {
    perror("pipe failed");
    exit(1);
  }
  if ((pid = fork()) < 0)
  {
    perror("fork failed");
    exit(2);
  }
  if (pid == 0)
  {
    close(pfd[1]);
    dup2(pfd[0], 0);
    close(pfd[0]);
    execlp("wc", "wc",
           (char *) 0);
    perror("wc failed");
    exit(3);
  } else {
    close(pfd[0]);
    dup2(pfd[1], 1);
    close(pfd[1]);
    execlp("ls", "ls",
           (char *) 0);
    perror("ls failed");
    exit(4);
  }
  exit(0);
}
```

Variations

Some common variations on this method of IPC are:

- A process may want to both write to and read from a child. In this case it creates two pipes. One of these is used by the parent for writing and by the child for reading. The other is used by the child for writing and the parent for reading.
- A pipeline may consist of three or more process (such as a C version of `ps | sed 1d | wc -l`). In this case there are lots of choices
    1. The parent can fork twice to give two children.
    2. The parent can fork once and the child can fork once, giving a parent, child and grandchild.
    3. The parent can create two pipes before any forking. After a fork therewill then be a total of 8 ends open (2 processes * two ends * 2 pipes). Most of these wil have to be closed to ensure that there ends up only one read and only one write end.
    4. As many ends as possible of a pipe may be closed before a fork. This minimises the number of closes that have to be done after forking.
