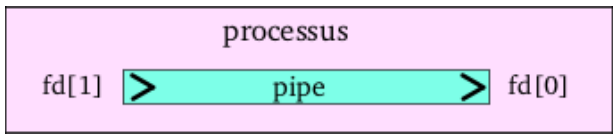
À à Á á Â â Ã ã Ä ä Ă ă Ă ă Å å Ā ā Ą ą Æ æ Ć ć Ç ç Č č Ĉ ĉ Ċ ċ Đ đ ð Ď ď È è
É é Ê ê Ë ë Ě ě Ē ē Ė ė Ę ę Ģ ģ Ĝ ĝ Ğ ğ Ġ ġ Ĥ ĥ Ì ì Í í Î î Ï ï Ǐ ǐ Ī ī İ ı Ļ ļ Ĵ ĵ Ķ
ķ Ĺ ĺ Ļ ļ Ľ ľ Ł ł Ŀ ŀ Ń ń Ñ ñ Ņ ņ Ň ň Ò ò Ó ó Ô ô Õ õ Ö ö Ǒ ǒ Ō ō Ő ő Œ œ Ø ø
Ŕ ŕ Ŗ ŗ Ř ř Ś ś Ş ş Š š Ŝ ŝ Ţ ţ Ť ť Ù ù Ú ú Û û Ü ü Ǔ ǔ Ŭ ŭ Ū ū Ů ů ű ú ǔ ù Ų ų
Ű ű Ŵ ŵ Ý ý Ÿ ÿ Ŷ ŷ Ź ź Ž ž Ż ż Þ þ ß Ħ ħ ¿ ¡ ç £ ¤ ¥ € ¦ § ª ¬ ¯ ° ± ÷ ‰ ¼ ½
¾ ¹ ² ³ µ ¶ † ‡ · • º ∀ ∂ ∃ ∂ ə ø ∇ ∈ ∉ ∋ ∏ ∑ ¯ − ∗ × ⁄ √ ∝ ∞ ∠ ∧ ∨ ∩ ∪ ∫ ∴
∼ ≅ ≈ ≠ ≡ ≤ ≥ ⊂ ⊃ ⊄ ⊆ ⊇ ⊕ ⊗ ⊥ · ◊ ℘ ℑ ℜ ℵ ♠ ♣ ♥ ♦ α β Γ γ Δ δ ε ζ η Θ θ ι
κ Λ λ µ Ξ ξ Π π ρ Σ σ Τ τ υ Φ φ χ Ψ ψ Ω ω ★ ☆ ☎ ☚ ☛ ☜ ☝ ☞ ☟ ☹ ☺ ✔ ✘ „ “ ”
, ‘ ’ « » ‹ › — – … ← ↑ → ↓ ↔ ⇐ ⇑ ⇒ ⇓ ⇔ © ™ ® ′ ″ [ ] { } ~ ( ) % § $ # | @

:

- Un pipe possède deux extremités.
- Il n'est possible de faire passer des informations que dans un sens unique. On peut donc écrire des informations à l'entré et en lire à la sortie.
- Les deux extrémités sont référencés par des descripteurs de fichiers (des entiers stoqués dans la variable fd).



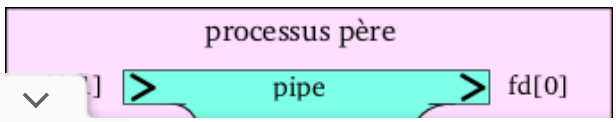
Concrétement, pour créer un simple pipe en langage C, voila ce qu'il faut écrire :

```
#include <stdio.h>
#include <memory.h>
#include <unistd.h>

int main( int argc, char ** argv )
{
 char buffer[BUFSIZ+1];

 /* create the pipe */
 int fd[2];
 pipe(fd);

 /* write into the pipe */
 write(fd[1], "Hello World\n", strlen("Hello World\n"));

 /* read the pipe and print the read value */
 read(fd[0], buffer, BUFSIZ);
 printf("%s", buffer);
}
```

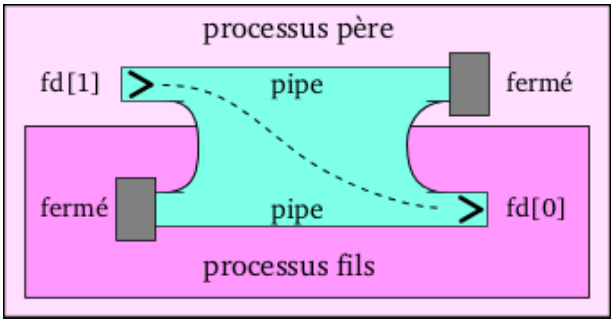
Nous utilisons la fonction "pipe(fd)" qui va réserver deux descripteurs de fichiers dans le tableau fd, "fd[0]" pour l'extrémité à lire et "fd[1]" pour l'extrémité à écrire.

Création d'un pipe dans un processus ayant un fils

La différence avec l'exemple précedent est que, en plus de créer un pipe, notre processus cré un fils. Le pipe est alors automatiquement partagé entre le père et le fils. Si l'un écrit dans le pipe alors on ne sait pas lequel des deux va recevoir l'information. Ceci peut donner des résultats inattendus.



Pour être certain de qui va écrire et qui va lire dans le pipe, il faut que les processus ferment les extrémités qu'ils n'utilisent pas.



De cette façon le processus père peut être certain que s'il écrit dans le pipe ("fd[1]"), le fils va reçevoir l'information en lecture ("fd[0]").

Concretement voici comment faire en C :

```
#include <stdio.h>
#include <memory.h>
#include <unistd.h>

int main( int argc, char ** argv )
{
 /* create the pipe */
 int  pfd[2];
 if (pipe(pfd) == -1)
   {
     printf("pipe failed\n");
     return 1;
   }

 /* create the child */
 int  pid;
 if ((pid = fork()) < 0)
   {
     printf("fork failed\n");
     return 2;
   }

 if (pid == 0)
   {
     /* child */
     char buffer[BUFSIZ];

     close(pfd[1]); /* close write side */

     /* read some data and print the result on screen */
     while (read(pfd[0], buffer, BUFSIZ) != 0)
       printf("child reads %s", buffer);

     close(pfd[0]); /* close the pipe */
   }
 else
   {
     /* parent */
     char buffer[BUFSIZ];

     close(pfd[0]); /* close read side */

     /* send some data into the pipe */
     strcpy(buffer, "HelloWorld\n");
     write(pfd[1], buffer, strlen(buffer)+1);

     close(pfd[1]); /* close the pipe */
   }

 return 0;
```

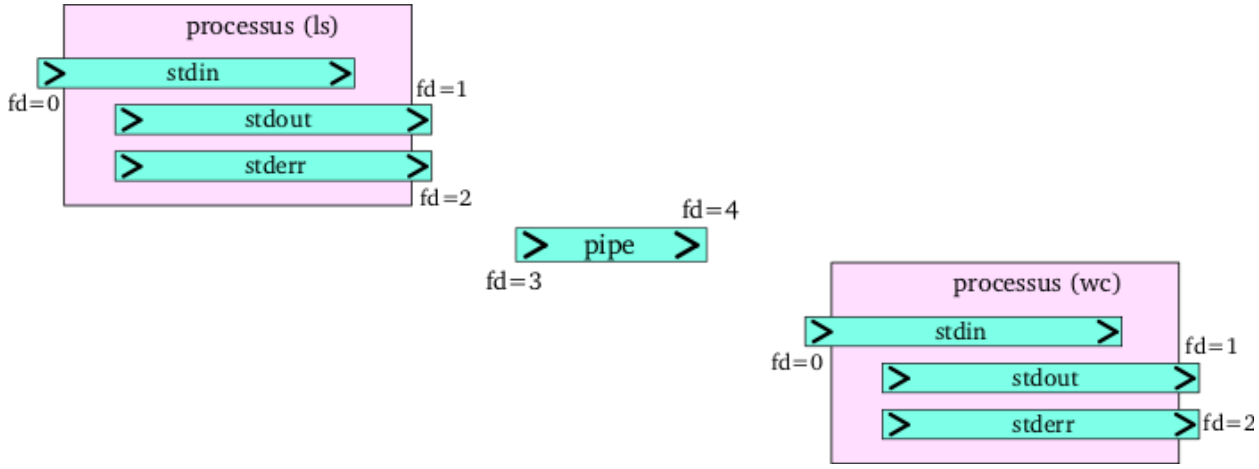
stdin, stdout, stderr

Ces trois pipes sont par défault créés dans chaques processus. Le premier, stdin, est branché par défaut sur l'entrée clavier tandis que stdout et stderr sont eux branchés sur la sortie écran. Des descripteurs de fichiers par défaut leur sont associés : 0 pour stdin, 1 pour stdout, et 3 pour stderr.

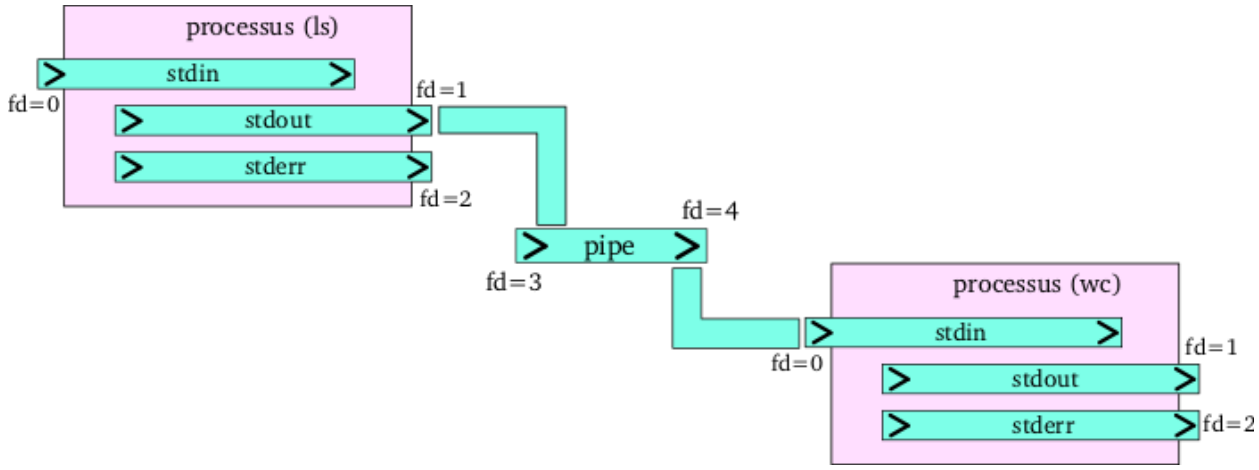
Maintenant que l'on connait le principe de communication par pipe, on est tenté de connecter ces pipes entre eux. Par exemple essayons de connecter stdout d'un premier processus avec le stdin d'un second. Cette opération est effectuée par le shell à chaque fois que deux commandes séparées par un pipe sont executées, par exemple pour relier la sortie stdout de la commande ls avec l'entrée stdin de la commande wc, il faut taper ceci dans un terminal : "ls | wc".

Pour réaliser ceci dans un programme en langage C, il faut procéder en plusieurs étapes.

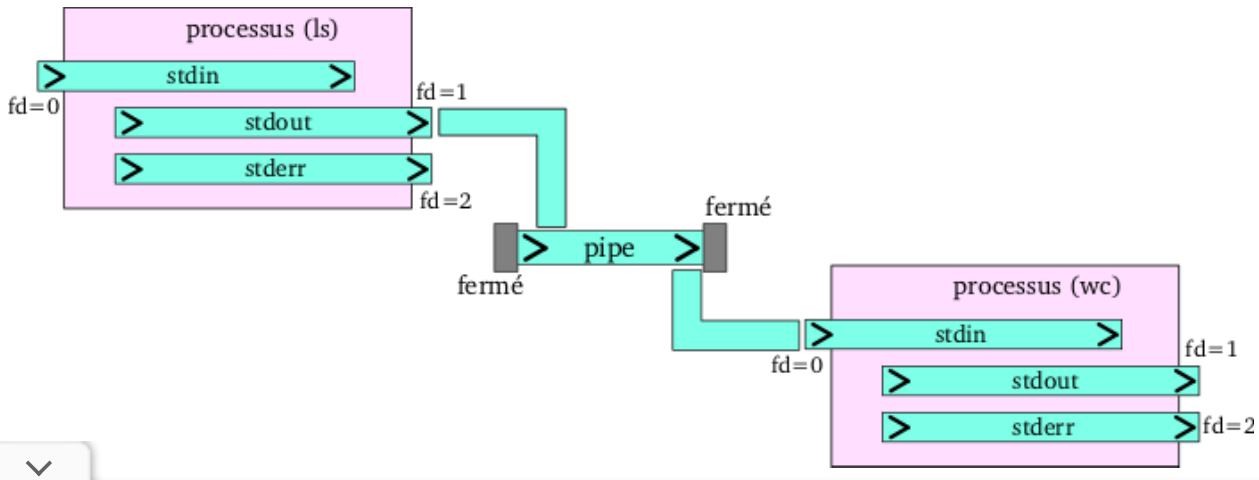
Il faut commencer par créer un pipe vide : "fd=3" en écriture et "fd=4" en lecture. On utilise donc la fonction "pipe(fd)" comme vue ci dessus.



Ensuite il faut que stdout du premier processus (fd1) soit connecté à l'entrée de notre pipe (fd3) et que la sortie de notre pipe (fd4) soit connecté à stdin de notre second processus (fd0). On appelera deux fois la fonction "dup2(param1, param2)" pour connecter fd1 à fd3 et fd4 à fd0. "dup2" prend en argument deux paramètres, ce sont des descripteurs de fichiers : param1 vaudra fd3 et param2 vaudra fd1 car on veut que fd3 soit assimilé (ou connecté) à fd1. Pour le second processus, param1 vaudra fd4 et parm2 vaudra fd0.



Finalement il faut fermer les extrémités de notre pipe pour éviter les comportements étranges, c'est un peu le même problème que dans la section précedente. On utilisera la commande "close(fd)" pour fermer les extremités de notre pipe.



```
  /* create the pipe */
  int pfd[2];
  if (pipe(pfd) == -1)
    {
      printf("pipe failed\n");
      return 1;
    }

  /* create the child */
  int pid;
  if ((pid = fork()) < 0)
    {
      printf("fork failed\n");
      return 2;
    }

  if (pid == 0)
    {
      /* child */
      close(pfd[1]); /* close the unused write side */
      dup2(pfd[0], 0); /* connect the read side with stdin */
      close(pfd[0]); /* close the read side */
      /* execute the process (wc command) */
      execlp("wc", "wc", (char *) 0);
      printf("wc failed"); /* if execlp returns, it's an error */
      return 3;
    }
  else
    {
      /* parent */
      close(pfd[0]); /* close the unused read side */
      dup2(pfd[1], 1); /* connect the write side with stdout */
      close(pfd[1]); /* close the write side */
      /* execute the process (ls command) */
      execlp("ls", "ls", (char *)0);
      printf("ls failed"); /* if execlp returns, it's an error */
      return 4;
    }
  return 0;
}
```

Ici on découvre l'utilisation d'une nouvelle fonction "execlp(…)" qui permet de remplacer le processus en cours par l'exécution d'une commande : dans notre cas la commande "wc" est appelée dans le processus fils et la commande "ls" est appelée dans le processus père. Cette fonction ne retourne rien sauf si une erreur se produit à l'exécution de la commande.

Conclusion

On a vu dans cet article comment créer des pipes mais on peut se demander à quoi peuvent ils servirent. Il y a certainement une infinité d'applications associées à ce mécanisme mais voici l'idée que j'avais derrière la tête et qui m'a poussé à étudier les pipes : créer une interface utilisateur (GUI) permettant de contrôler un programme en ligne de commande (qui utilise donc stdin, stdout et stderr pour communiquer avec l'extérieur). En effet, si on peut contrôler l'entrée (stdin) et la sortie (stdout, stderr) d'un programme en l'encapsulant dans un autre programme père, on peut tout faire avec !

Sources

- http://www.cryptnet.net/fdp/gtk/exe... [http://www.cryptnet.net/fdp/gtk/exec.html]
- http://pandonia.canberra.edu.au/OS/... [http://pandonia.canberra.edu.au/OS/l9_1.html]
- http://www.gidforums.com/t-3369.html [http://www.gidforums.com/t-3369.html]
- http://mkssoftware.com/docs/man3/ex... [http://mkssoftware.com/docs/man3/execl.3.asp]
- http://www.opengroup.org/onlinepubs... [http://www.opengroup.org/onlinepubs/009695399/functions/dup.html]