Pipe

- Objet : Le pipe
- Niveau requis :
  avisé
- Commentaires : *Utiliser le caractère pipe.« | ».*
- Suivi :
  à-tester
  - Création par e-miel le 20-04-2011 18:31:31
  - Testé par <...> le <...>
- Commentaires sur le forum : Lien vers le forum concernant ce tuto[1)]

Introduction

Le caractère pipe est « | » et s'obtient avec la combinaison de touches : Alt Gr avec le 6 du clavier des **lettres**.

Les redirections d'entrée/sortie[2)] permettent de rediriger les résultats vers un fichier.
Ce fichier peut ensuite être réinjecté dans un filtre pour en extraire d'autres résultats.
Cela oblige à taper deux lignes :

1. une pour la redirection vers un fichier,
2. l'autre pour rediriger ce fichier vers le filtre.

Les tubes ou pipes permettent de rediriger directement le canal de sortie → d'une commande vers → le canal d'entrée d'autre.

Ainsi, les 2 redirections suivantes :

```
ls -l > resultat.txt
```

Puis :

```
wc < resultat.txt
```

Deviennent cette commande unique :

```
ls -l | wc
```

Il est possible de placer plusieurs « | » sur une même ligne.</code>

```
ls -l | wc | wc
```

```
      1       3      24
```

La première commande n'est pas forcément un filtre. L'essentiel est qu'un résultat soit délivré.
Idem pour la dernière commande qui peut par exemple être une commande d'une édition ou d'impression.

Enfin, la dernière commande peut elle-même faire l'objet d'une redirection en sortie.

```
ls -l | wc > resultat.txt
```

Bien que ce ne soit pas un problème de mettre un espace à la suite d'un **pipe**, pourvu que ce soit bien un espace et pas autre chose.
En fait, ce n'est pas l'espace qui peut introduire une erreur d'interprétation, mais par exemple, si tu tapes *un espace insécable*, pour le shell cet *espace insécable* n'est pas un délimiteur de mot et donc il analyse l'*espace insécable* représenté de manière non visible comme faisant partie du mot qui le suit ce qui peut engendrer une erreur de sortie.

Par sécurité, nous pouvons donc rédiger les commandes utilisant **pipe** sans espace dans la redirection.
Exemple :

```
commande1 |commande2 |Tchibâââ
```

Remarque :
Ce qui vaut pour le pipe |, vaut également pour la redirection > amha

Fonctionnement en détail

Un *pipe* permet à deux processus de s'échanger des données. Les commandes Shell permettent de créer facilement un *pipe* :

```
envoyeur | receveur
```

Dans cet exemple, la sortie standard de **envoyeur** est reliée à l'entrée standard de **receveur**.
Le Shell est un outil très pratique pour créer un *pipe*, mais en aucun cas les données du *pipe* ne transiteront par le Shell lors de l'exécution.
Un *pipe* est aussi appelé FIFO (First In First Out) cela signifie que les données qui sortiront en premier de **envoyeur** seront également les données qui rentreront en premier dans **receveur**.

Cependant, l'ordre est celui dans lequel les appels-système à WRITE et READ ont été invoqués, pas l'ordre dans lequel les processus tentent de les réaliser.

Par défaut, les *pipe* sont bloquants, ce qui signifie que lorsque **envoyeur** écrit dans le *pipe*, il s'endort jusqu'à ce que le processus **receveur** ait terminé de TOUT lire, ce qui réveille **envoyeur**.
Si **receveur** est prêt à lire dans le *pipe* mais qu'**envoyeur** n'y a encore rien écrit, **receveur** s'endort et se réveillera dès qu'il y aura quelque chose à lire.

Lorsque vous programmez une application, vous pouvez choisir de rendre les *pipes* non-bloquants, grâce au flag O_NDELAY ou O_NONBLOCK.
Un *pipe* peut très bien être bloquant d'un côté et non-bloquant de l'autre côté.
L'ordre de réception des données peut différer suivant que le *pipe* soit bloquant (par défaut) ou non-

bloquant (flag O_NDELAY).

Testons le pipe en pratique

Cette partie technique ne concerne que des personnes très avisées ! 😄

Afin que vous puissiez expérimenter par vous-même le fonctionnement d'un *pipe*, je vous ai préparé deux commandes: **rd** (pour READ) et **wr** (pour WRITE) dont voici le code source **src.c**:

```
# include <fcntl.h>
# include <stdio.h>
# include <string.h>
# include <sys/wait.h>
# include <time.h>
# include <unistd.h>

char buffer[100000000] ;

int main(int n, char **v)
{
    if( n==1 || strcmp(v[1],"--help")==0 )
    {
        fprintf( stderr, "Usage: %s [-n] <timeout>s<length>k ...\n", v[0] )
;
        return 1 ;
    }

    int i = 1, fd = ( strcmp(v[0],"wr") == 0 ) ? 1 : 0 ;

    if( strcmp(v[i],"-n") == 0 )
    {
        fcntl( fd, F_SETFL, O_NDELAY ) ;
        i++ ;
    }

    struct timespec t0, t1 ;
    clock_gettime( CLOCK_REALTIME, &t0 ) ;

    for(  ; i<n ; i++ )  if( fork() == 0 )
    {
        char *args = v[i] ;
        long timeout, length ;
        sscanf( args, "%li", &timeout ) ;
        args = index( args, 's' ) + 1 ;
        sleep( timeout ) ;
        while( *args )
        {
```

```
            sscanf( args, "%li", &length ) ;
            args = index( args, 'k' ) + 1 ;
            long ret = fd ? write(1,buffer,length*1024) :
read(0,buffer,length*1024) ;
            clock_gettime( CLOCK_REALTIME, &t1 ) ;
            long t = (t1.tv_sec - t0.tv_sec)*10 + (t1.tv_nsec -
t0.tv_nsec)/100000000 ;
            fprintf( stderr, fd ? "\033[%im(%lis)%2li.%lis %5lik >\033[0m\n"
: "\033[%im(%lis)%2li.%lis        >%5lik\033[0m\n", fd==0 && ret>0, timeout,
t/10, t%10, (ret+1023)/1024 ) ;
        }
        return 0 ;
    }

    signal( SIGCHLD, SIG_IGN ) ;
    wait( NULL ) ;

    clock_gettime( CLOCK_REALTIME, &t1 ) ;
    long t = (t1.tv_sec - t0.tv_sec)*10 + (t1.tv_nsec -
t0.tv_nsec)/100000000 ;
    fprintf( stderr, fd ? "    %2li.%lis  CLOSE >\n" : "    %2li.%lis
> CLOSE\n", t/10, t%10 ) ;

    return 0 ;
}
```

Compilez-le avec :

```
gcc -s -lrt -Wall -std=gnu99 -o rd src.c
```

```
ln -s rd wr
```

À présent, vous vous retrouvez avec deux commandes supplémentaires :

- **rd** qui réalise des READ sur son entrée standard avec les retards (secondes) et les tailles (ko) de votre choix.
- **wr** qui réalise des WRITE sur sa sortie standard avec les retards (secondes) et les tailles (ko) de votre choix.

Vous pouvez écrire **rd** et **wr** plutôt que **./rd** et **./wr** grâce à :

```
PATH="$PATH:."
```

Scénarios simples

Pour commencer, voici 6 scénarios simples, pour faire comprendre par l'exemple comment réagit un *pipe*:

Exemple 1

Un cas d'école, non-représentatif de la complexité des véritables applications :

```
wr 3s30k | rd 2s30k
```

Dans cet exemple, **wr** et **rd** sont lancés simultanément et reliés grâce à un *pipe* créé par le Shell. **rd** va attendre 2 secondes puis tenter de lire 30ko dans le *pipe*, mais comme il n'y a encore rien à lire, **rd** va être mis en sommeil. **wr** attend 3 secondes puis écrit 30ko dans le *pipe*, ce qui va réveiller **rd** (qui aura dormi 1 seconde) avec les données demandées. Dans cet exemple, la copie s'est faite directement de la mémoire virtuelle de **wr** à la mémoire virtuelle de **rd** sans transiter par aucun tampon intermédiaire. Les processus **wr** et **rd** se réveillent une fois que Linux a copié les 30ko de données.

Cet exemple est naïf, car les processus envoyeur et receveur ne sont pas censés utiliser “comme par hasard” des paquets de même taille (ici 30ko) pour transférer les données.

Exemple 2

Tampon automatique, **wr** ne se rend compte de rien :

```
wr 2s30k | rd 3s20k
```

Dans cet exemple, **wr** attend 2 secondes puis tente d'écrire 30ko dans le *pipe*. À ce moment, **rd** n'a pas encore fait son READ, donc où vont les 30ko? Linux les stocke provisoirement dans un tampon limité à 16 pages mémoire (sachant qu'une page fait 4ko, le quota par défaut du tampon est de 64ko) en attendant que **rd** veuille bien lire son entrée standard. Le processus **wr** se termine. Une seconde plus tard, **rd** va lire les 20 premiers ko du tampon, sans savoir qu'il n'y a plus personne derrière son entrée standard. Puis **rd** se termine, et Linux va jeter les 10ko non-récupérés.

fcntl() permet de grossir le tampon du *pipe* jusqu'à 1Mo (ou plus avec l'intervention du root). Cependant, dans le cas de gros transferts, il est plus efficace de réaliser de grosses copies directes avec des WRITE et READ bloquants, plutôt que de multiplier le nombre de copies pour éviter de dormir... et de toute façon, tampon ou pas, un processus dort TOUJOURS pendant une copie, que cette copie soit bloquante ou non.

Exemple 3

Comme l'exemple 2, mais avec un zéro en plus dans les tailles, ça change tout :

```
wr 2s300k | rd 3s200k
```

wr tente d'écrire 300ko dans le *pipe* et **rd** n'a pas encore fait son READ. Comme le tampon du *pipe*

n'est que de 64ko, les données seront donc copiées en prise directe: **wr** s'endort jusqu'à ce que **rd** finisse de récupérer les 300ko. Or **rd** ne lit que 200ko, **wr** reste endormi en attendant que **rd** fasse un autre READ, seulement contre toute attente, **rd** se termine, ce qui laisse les données de **wr** en plan. Ne sachant pas gérer la situation, **wr** se tue.

Dans l'exemple 2, **wr** ne se rendait pas compte que **rd** n'allait pas récupérer toutes les données, sauf qu'ici le tampon est trop petit et **wr** est donc en prise directe avec **rd**. Grâce à signal(), il aurait été possible de faire réagir **wr** autrement qu'en se suicidant à la réception d'un SIGPIPE, mais cela ne change rien au problème: tout processus receveur est censé attendre la fin du flux de données avant de se terminer. Dans les exemples 2 et 3, **rd** est malpoli.

Exemple 4

Ça devient intéressant, et commence à ressembler à la réalité :

```
wr 2s300k | rd 3s200k 4s200
```

wr tente d'écrire 300ko dans le *pipe*, **rd** récupère 200ko sans utiliser le tampon, puis une seconde plus tard, **rd** tente de récupérer 200ko mais reçoit les 100ko restants, ce qui réveille **wr**. Les 2 processus n'ont plus rien à faire, et se terminent donc simultanément.

Dans cet exemple, aucun problème.

Exemple 5

Comme l'exemple 4, mais l'écriture est non-bloquante :

```
wr -n 2s300k | rd 3s200k 4s200
```

wr (avec l'option **-n**) tente d'écrire 300ko, mais personne n'attend à l'autre bout du *pipe*. **wr** va donc remplir les 64ko du tampon du *pipe*. Une seconde plus tard, **rd** tente de lire 200ko et récupère les 64ko stockés dans le tampon. Une seconde plus tard, **rd** tente à nouveau de lire 200ko mais ne récupère rien, car le tampon est vide. **rd** va donc se terminer en ayant récupéré 64ko au lieu de 300ko.

L'écriture non-bloquante a rendu cet exemple naïf, mais rassurez-vous: “écriture non-bloquante” n'est pas nécessairement synonyme de “pertes de données”, car on peut s'en sortir avec une boucle, mais, cela consommera beaucoup de ressource CPU et sera de toute façon moins efficace qu'un processus qui dort, car un processus qui dort se réveille toujours au bon moment.

Exemple 6

Comme l'exemple 1, mais la lecture est non-bloquante :

```
wr 3s30k | rd -n 2s30k
```

rd (avec l'option **-n**) tente de lire 30ko, mais il n'y a rien à lire. Au lieu de s'endormir en attendant que **wr** fasse un WRITE, **rd** va tout simplement passer à la suite, c'est-à-dire se terminer. Une seconde plus tard, **wr** va vouloir écrire 30ko, mais comme il n'y a personne à l'autre bout du *pipe*, **wr** va recevoir un SIGPIPE et se suicider.

L'exemple 1, qui fonctionnait parfaitement, échoue si l'on rend la lecture non-bloquante.

Scénario complexe

Voici comment les données se comportent suivant que les WRITE et les READ soient bloquants ou non-bloquants. On constate non seulement qu'il y a pertes de données, mais qu'en plus les données n'arrivent pas forcément dans le même ordre.

Les temps demandés par l'utilisateur (entre parenthèses) ne sont pas les mêmes que ceux auxquels les primitives système ont été exécutées. De plus, les lignes correspondant à des transferts achevés s'affichent en surbrillance dans une console. Amusez-vous bien. 😉

WRITE > READ

```
wr 1s10k300k7k 2s5k 4s200k | rd 0s2k 3s5000k 5s6k
```

retour de la commande

```
  (1s) 1.0s    10k >
  (0s) 1.0s        >    2k
  (2s) 3.0s     5k >
  (1s) 3.0s   300k >
  (1s) 3.0s     7k >
  (3s) 3.0s        >  320k
  (5s) 5.0s        >    6k
       5.0s        > CLOSE
       5.0s  CLOSE >
```

Il se passe 2 secondes entre la dernière action de **wr** et la fermeture du *pipe*, **wr** a donc été tué durant son sommeil.

WRITE non-bloquant > READ

```
wr -n 1s10k300k7k 2s5k 4s200k | rd 0s2k 3s5000k 5s6k
```

retour de la commande

```
  (1s) 1.0s    10k >
  (1s) 1.0s    52k >
  (1s) 1.0s     0k >
  (0s) 1.0s        >    2k
  (2s) 2.0s     0k >
  (3s) 3.0s        >   60k
  (4s) 4.0s    64k >
       4.0s  CLOSE >
  (5s) 5.0s        >    6k
       5.0s        > CLOSE
```

La majorité des données a été perdue, alors que le scénario est identique au premier, au type d'écriture près.

WRITE > READ non-bloquant

```
wr 1s10k300k7k 2s5k 4s200k | rd -n 0s2k 3s5000k 5s6k
```

retour de la commande

```
  (0s) 0.0s        >    0k
  (1s) 1.0s    10k >
  (2s) 3.0s     5k >
  (1s) 3.0s   300k >
  (1s) 3.0s     7k >
  (3s) 3.0s        >  322k
  (5s) 5.0s        >    6k
       5.0s        > CLOSE
       5.0s  CLOSE >
```

Comme précédemment, on voit que **wr** a été tué durant son sommeil.

WRITE non-bloquant > READ non-bloquant

```
wr -n 1s10k300k7k 2s5k 4s200k | rd -n 0s2k 3s5000k 5s6k
```

retour de la commande

```
  (0s) 0.0s         >    0k
  (1s) 1.0s    10k >
  (1s) 1.0s    52k >
  (1s) 1.0s     0k >
  (2s) 2.0s     0k >
  (3s) 3.0s         >   62k
  (4s) 4.0s    64k >
       4.0s  CLOSE >
  (5s) 5.0s         >    6k
       5.0s         > CLOSE
```

Quasiment toutes les données ont été perdues.

Conclusion

- **Pipe bloquant** = comportement par défaut = aucune consommation CPU quand le processus dort = se réveille toujours au bon moment 😀
- **Pipe non bloquant** = problèmes de données perdues et consommation CPU 😕

L'immense majorité des programmes ne personnalisent pas le comportement de leurs descripteurs, donc si vous utilisez des pipes dans vos scripts, ils seront bloquants :

- Les échanges de *petite taille* : texte brut échangé entre **awk**, **sed**, **bash**, **latex**, et en général tout traitement de script par **python** (choisi pour son efficacité à traiter les expressions régulières) se font implicitement au travers du tampon de 64ko, c'est la solution la plus rapide : chacun des processus participants enchaîne ses nombreux petits traitements internes sans s'arrêter puis se termine indépendamment de l'autre processus.
- Les échanges de *grande taille* : vidéo brute échangée entre **ffmpeg**, **x264**, et en général toute compression de grande taille par **gzip**, **xz**, se fait implicitement sans tampon, copie directe de processus à processus durant leur sommeil, c'est la solution la plus rapide : le tampon n'est jamais utilisé même partiellement.

La quantité de données échangées déterminera si le comportement est **tampon** ou **sommeil**, mais dans les deux cas, les pipes sont bloquants.

1)

N'hésitez pas à y faire part de vos remarques, succès, améliorations ou échecs !
[2)]

Les redirections en commande