

# Les IPC : Inter-Process Communication

**D'après :**  
**Programmation multitâche sous**  
**unix**  
*et autres sources*

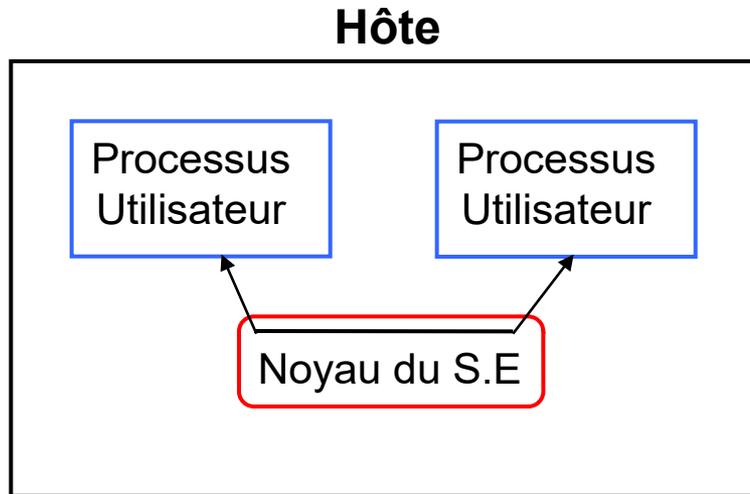
Groupe Isaip-Esaip  
P. Trégouët – A. Schaal

# Besoins de communication entre processus

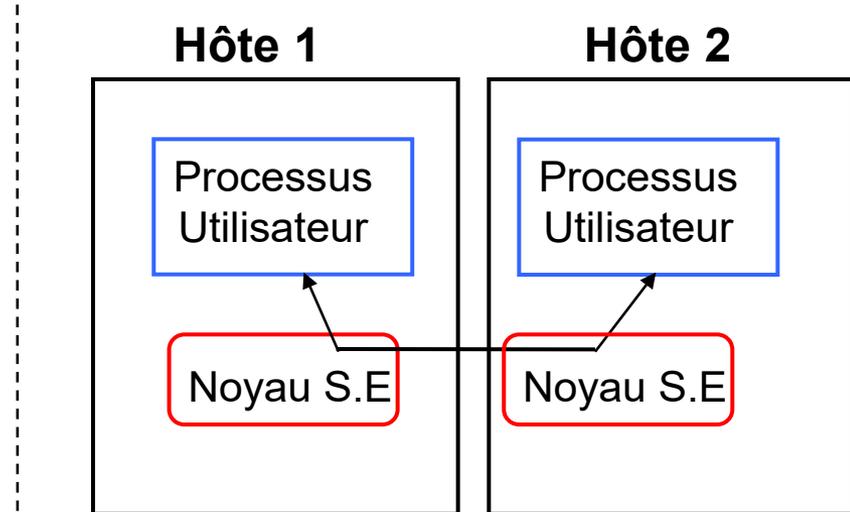
- Deux processus peuvent éventuellement partager le même code (ceci est sous le contrôle du système) mais les espaces mémoires réservés aux variables sont complètement étanches : un processus n'a aucun accès aux variables d'un autre processus
- On peut néanmoins créer une application multitâche et avoir besoin de **synchroniser ces tâches** ou **d'échanger des informations entre ces tâches**. Les systèmes d'exploitations sont donc munis de mécanismes appropriés
  - Synchronisation par **signaux** ou **sémaphores**
  - Communication de données par **fichiers**, **tubes** (pipes) files de **messages** (queues ou boîtes à lettres) variables implantées dans un **segment de mémoire partagée**

- outils de communication
  - famille des signaux
    - signaux
  - famille des files de messages
    - files de messages
    - tubes
  - mémoire partagée
- outils de synchronisation
  - famille des sémaphores
    - sémaphores binaire et à compte
    - mutexes

# Communications locales ou distantes



Communication intra-système



Communication inter-systèmes



## La communication par signaux

- Un signal peut être assimilé à une interruption logicielle (par analogie avec les interruptions matérielles)
- Comme pour une interruption matérielle, la réception d'un signal interrompt le traitement en cours et **exécute automatiquement la fonction associée au signal (programmation événementielle)**.
- En langage C, l'association entre le numéro du signal et la fonction est réalisé par un appel à la fonction système **signal()**.

Note : la fonction **sigaction()** peut être utilisée à la place de **signal()**.  
Voir **man sigaction**.

# La communication par signaux

- ***Un signal est***
  - Envoyé par un processus
  - Reçu par un autre processus (éventuellement le même)
  - Véhiculé par le noyau
- ***Comment réagit un processus qui reçoit un signal ?***
  - Il interrompt le traitement en cours
  - il exécute la fonction de traitement du signal
  - il reprend l'exécution du traitement interrompu
  - ***Si le processus était endormi, il est réveillé par le signal. Après l'exécution de la fonction associée au signal, dans le cas où il est réveillé avant la fin d'une temporisation il ne se rendort pas; par contre s'il attendait la fin d'une entrée-sortie, il continue à attendre.***
- ***Comportement associée à la réception d'un signal :***
  - En général un comportement **par défaut** est défini : Terminaison anormale du processus (ex: violation de mémoire, division par zéro)
  - **Le processus peut (parfois) ignorer le signal**
  - Le processus peut redéfinir, par une **fonction spécifique**, son comportement à la réception d'un signal

# Origine des signaux

- Frappe de caractères

touche	signal
CTRL-C	SIGINT
CTRL-\	SIGQUIT
CTRL-Z	SIGTSTP
CTRL-S	SIGSTOP
CTRL-Q	SIGCONT

- Erreurs de programme
  - Violation de mémoire, SIGSEGV
  - Division par zéro, SIGFPE
  - ...
- Commande kill  
Primitive système : `kill()`, `alarm()`.

## En shell

```
$ kill -s KILL 1984  
$ kill -KILL 1984  
$ kill -9 1984
```

Toutes ces commandes envoient le signal non masquable SIGKILL au processus de PID 1984

# Liste des signaux en shell

**\$ kill -l**

1) SIGHUP	2) SIGINT	3) SIGQUIT	4) SIGILL
5) SIGTRAP	6) SIGABRT	7) SIGBUS	8) SIGFPE
9) SIGKILL	10) SIGUSR1	11) SIGSEGV	12) SIGUSR2
13) SIGPIPE	14) SIGALRM	15) SIGTERM	17) SIGCHLD
18) SIGCONT	19) SIGSTOP	20) SIGTSTP	21) SIGTTIN
22) SIGTTOU	23) SIGURG	24) SIGXCPU	25) SIGXFSZ
26) SIGVTALRM	27) SIGPROF	28) SIGWINCH	29) SIGIO
30) SIGPWR	31) SIGSYS	33) SIGRTMIN	34) SIGRTMIN+1
35) SIGRTMIN+2	36) SIGRTMIN+3	37) SIGRTMIN+4	38) SIGRTMIN+5
39) SIGRTMIN+6	40) SIGRTMIN+7	41) SIGRTMIN+8	42) SIGRTMIN+9
43) SIGRTMIN+10	44) SIGRTMIN+11	45) SIGRTMIN+12	46) SIGRTMIN+13
47) SIGRTMIN+14	48) SIGRTMIN+15	49) SIGRTMAX-15	50) SIGRTMAX-14
51) SIGRTMAX-13	52) SIGRTMAX-12	53) SIGRTMAX-11	54) SIGRTMAX-10
55) SIGRTMAX-9	56) SIGRTMAX-8	57) SIGRTMAX-7	58) SIGRTMAX-6
59) SIGRTMAX-5	60) SIGRTMAX-4	61) SIGRTMAX-3	62) SIGRTMAX-2
63) SIGRTMAX-1	64) SIGRTMAX		

# Signaux les plus fréquemment utilisés

1	SIGHUP	terminaison du processus leader
2	SIGINT	frappe d'interruption (CTRL-C)
3	SIGQUIT	frappe de quit (CTRL-\)
4	SIGILL	instruction illégale
6	SIGABRT	problème matériel
7	SIGBUS	erreur sur le BUS
8	SIGFPE	erreur arithmétique
9	SIGKILL	signal de terminaison la réaction à ce signal ne peut être redéfinie
10	SIGUSR1	signal utilisateur
11	SIGSEGV	violation écriture mémoire
12	SIGUSR2	signal utilisateur
13	SIGPIPE	écriture sur un tube non ouvert en lecture
14	SIGALRM	fin de temporisateur
15	SIGTERM	terminaison normale d'un processus
17	SIGCHLD	terminaison (arrêt) d'un fils
18	SIGCONT	continuation d'un processus arrêté par SIGSTOP
19	SIGSTOP	signal de suspension d'exécution de processus la réaction à ce signal ne peut être redéfinie
20	SIGTSTP	frappe du caractère de suspension sur le clavier (CTRL-Z)

# Envoi de signaux en langage C

```
#include <signal.h>
```

```
int kill(pid_t pid,int sig)
```

pid : > 0 : pid de processus destinataire

sig : le signal à envoyer

Envoie un signal à un processus

## NAME

kill — send a signal to a process or a group of processes

## SYNOPSIS

```
#include <signal.h>
```

```
int kill(pid_t pid, int sig);
```

## DESCRIPTION

The kill() function shall send a signal to a process or a group of processes specified by pid. The signal to be sent is specified by sig and is either one from the list given in <signal.h> or 0. If sig is 0 (the null signal), error checking is performed but no signal is actually sent. The null signal can be used to check the validity of pid.

For a process to have permission to send a signal to a process designated by pid, unless the sending process has appropriate privileges, ...

# Envoi de signaux en langage C

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

```
unsigned alarm (unsigned nb)
```

Envoie un signal **SIGALRM** au processus en cours dans **nb** secondes.

Attention :

ne pas mélanger l'utilisation de alarm et de sleep  
qui partagent le même "timer" (voir man alarm)

## NAME

alarm — schedule an alarm signal

## SYNOPSIS

```
#include <unistd.h>  
unsigned alarm(unsigned seconds);
```

## DESCRIPTION

The alarm() function shall cause the system to generate a SIGALRM signal for the process after the number of realtime seconds specified by seconds have elapsed. Processor scheduling delays may prevent the process from handling the signal as soon as it is generated.

If seconds is 0, a pending alarm request, if any, is canceled.

Alarm requests are not stacked; only one SIGALRM generation can be scheduled in this manner...

# Préparer la réception des signaux en C

```
#include <signal.h>
```

```
void (*signal(int signum, void (*handler)(int)))(int);
```

signum : le signal à intercepter

handler : pointeur sur une fonction qui gère le signal

c'est simplement en C le nom de cette fonction

ou constante correspondant à un comportement prédéfini:

SIG\_DFL : comportement par défaut (terminaison)

SIG\_IGN : ignorer le signal

signal() renvoie la valeur précédente du gestionnaire de signaux, ou SIG\_ERR en cas d'erreur.

---

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

```
int pause(void);
```

Endort le processus appelant jusqu'à ce qu'il reçoive un signal.

renvoie toujours -1, et errno est positionné à la valeur EINTR.

on peut aussi utiliser **sleep()** si on veut limiter l'attente

## Sommeil de processus

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

```
unsigned int sleep (unsigned int nb_secondes)
```

endort le thread appelant jusqu'à ce qu'un nombre de secondes réelles se soient écoulées ou jusqu'à ce qu'un signal non ignoré soit reçu.

Un **signal** reçu interrompt le sommeil du processus.

Le programme continue alors son exécution après l'appel de sleep().

La valeur retournée par sleep() est le nombre de secondes restantes

# Communication par signaux : exemple

```
#include<signal.h>

void standard(int numero) {
    printf("standard Signal %d recu : \n",numero);
}
void message(int numero) {
    printf("message Signal %d recu\n",numero);
}

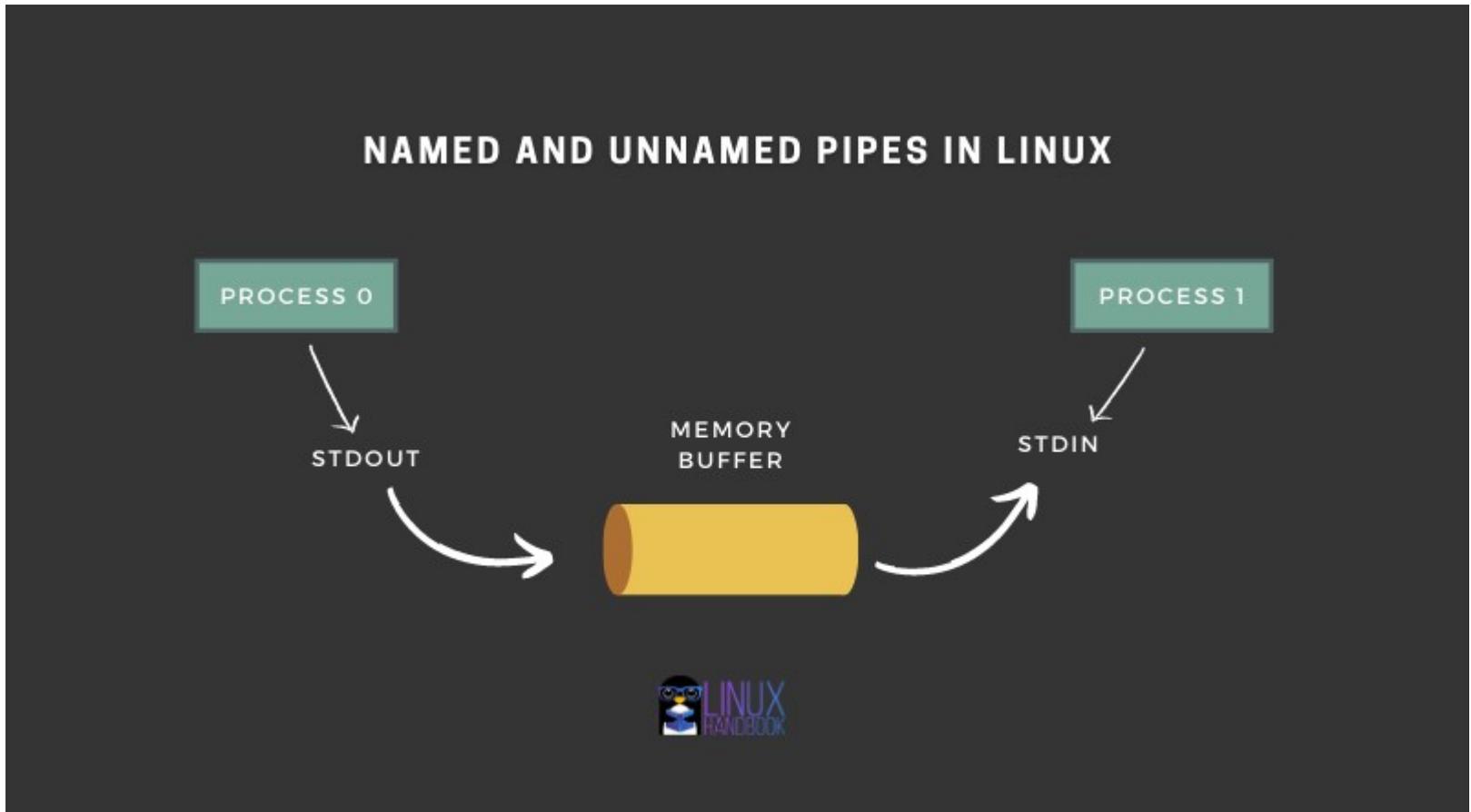
main() {
int numero;
    // traitements spécifiques
    for (numero=1;numero<32;numero++)signal(numero,standard);
    signal(SIGINT,message);  signal(SIGQUIT,message);
    //traitements prédéfinis
    signal(1,SIG_DFL);  signal(4,SIG_IGN);

    // Rien à faire, sauf réagir aux signaux: attente passive
    while (1) pause();
}
```

Remarque sous Linux, « signal » est fiable :

- pendant l'exécution du gestionnaire le signal correspondant est bloqué
- le gestionnaire reste en place à l'issue de son exécution.
- Pour plus d'options : [sigaction](#)

# Pipes (tubes)



# Tubes : principe

- Un tube (**anonyme**) est traité comme un fichier ouvert
- Il est transmis aux processus enfants de la même manière que les fichiers ouverts
- Un tube anonyme est cependant un fichier particulier :
  - Il n'a **pas de nom dans l'arborescence des fichiers**
  - Il est connu par **deux descripteurs** (lecture, écriture)
- On lit/écrit dans un tube  
comme on lit/écrit dans un fichier séquentiel
  - Utilisation des fonctions read et write  
(éventuellement fscanf, fprintf etc...)
  - Les données ne sont pas structurées:  
le pipe ne contient qu'une suite d'octets
- Un tube est comme un "tuyau" avec une entrée et une sortie :
  - Plusieurs processus peuvent y écrire des données
  - Plusieurs processus peuvent y lire des données

# Tubes : principe

- Le fonctionnement est de type **FIFO** :
  - On lit toujours les octets dans l'ordre où ils ont été déposés dans le tube
  - Il n'est pas possible de lire un octet sans voir lu tous ceux qui le précèdent
  - Une donnée lue par un processus ne pourra jamais être relue par un autre processus
- Les droits d'accès sont ceux des droits d'accès Unix sur les fichiers (rwx, rwx, rwx)
- La lecture est bloquante
- Lecteur bloqué si tube vide mais ouvert en écriture par un autre processus
- Si un tube est vide et si tous les descripteurs susceptibles d'y écrire sont fermés, la primitive **read()** renvoie la valeur 0 (fin de fichier atteinte).
- Un processus qui écrit dans un tube qui n'a plus de lecteurs (tous les descripteurs susceptibles de lire sont fermés) reçoit le signal **SIGPIPE**

# TUBES : primitives utiles

- Création :

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

```
int pipe(int p_desc[2])
```

p\_desc[0] pour la lecture, p\_desc[1] pour l'écriture

- Entrées/Sorties :
  - write (p\_desc[1], buf\_ecrire, nbre\_octets)
  - read (p\_desc[0], buf\_lire, nbre\_octets)
  - FILE \* fdopen():fprintf(), fscanff(),etc...
  - fflush : écrire la zone tampon dans le tube avant qu'elle ne soit remplie
- Pour éviter l'interblocage fermer le côté de tube non utilisé, par la primitive **close()**

# interblocage

## Utilisation typique d'un tube

- un processus écrivain
- un processus lecteur

## Nombre de lecteurs

- nombre de descripteurs associés à la lecture depuis le tube
- si nul, le tube est inutilisable, les écrivains pourront être prévenus

## Nombre d'écrivains

- nombre de descripteurs associés à l'écriture dans le tube
- si nul, le tube vide est inutilisable, les lecteurs pourront être prévenus

Bonne pratique :

systematiquement fermer, au plus tôt, tous les descripteurs non utilisés : garder un descripteur en écriture sur un tube = **potentiellement bloquer un processus**

## TUBES : exemple

```
#include <signal.h>

main(){
int i, ret, p_desc[2];
char c;
pipe(p_desc);
write(p_desc[1], "AB", 2);
close(p_desc[1]);
for (i=1; i<=4; i++) {
    ret = read(p_desc[0], &c, 1);
    if (ret == 1) printf("valeur lue: %c\n", c);
    else perror("impossible de lire dans le tube\n");
}
}
```

### Résultat de l'exécution :

valeur lue : A  
valeur lue: B  
impossible de lire dans le tube  
impossible de lire dans le tube

## TUBES : interblocages, étreintes fatales

```
int main () {
    int fdts[2], /* to son */
    fdfs[2]; /* from son */
    char bufr[BSIZE], bufw[BSIZE];
    pipe(fdts);
    pipe(fdfs);
    if(fork()) { /* pere */
        read(fdts[0], bufr, 1);
        write(fdfs[1], bufw, 1);
    } else { /* fils */
        read(fdts[0], bufr, 1);
        write(fdfs[1], bufw, 1);
    }
    printf("bye\n");
    exit(EXIT_SUCCESS);
}
```

Indéblocable  
(tube anonyme!)

# TUBES Nommés

Les tubes ordinaires, sont **non visibles dans l'arborescence des fichiers**, et accessibles uniquement aux processus d'une **même affiliation**

- Tubes nommés : **visibles dans l'arborescence de fichiers**, utilisables entre processus n'ayant pas de parents communs
- Aucune information n'est enregistrée sur disque, le transfert d'information s'effectue toujours en mémoire.
- Primitives correspondantes :  
    int mknod(const char \*nom\_fich, mode\_t mode, dev\_t dev)  
    **int mkfifo (const char \**pathname*, mode\_t *mode*);**
- Ouverture de tube par les processus connaissant le nom du tube :
  - En lecture/écriture par **open()**
  - Par défaut, ouverture bloquante :  
lecteurs et écrivains s'attendent = synchronisation

# Les tubes nommés: commande mkfifo

```
$ mkfifo mypipe
```

Affichage des attributs du tube créé

```
$ ls -l mypipe
```

```
prw----- 1 username grname 0 sep 12 11:10 mypipe
```

Modification des permissions d'accès (comme un fichier ordinaire)

```
$ chmod g+rw mypipe
```

```
$ ls -l mypipe
```

```
prw-rw---- 1 username grname 0 sep 12 11:12 mypipe
```

Remarque : **p** indique que c'est un tube.

- Une fois le tube créé, il peut être utilisé pour réaliser la communication entre deux processus.
- Chacun des deux processus ouvre le tube, l'un en mode écriture et l'autre en mode lecture.

```
$ echo "salut" > mypipe
```

```
$ cat mypipe
```

```
salut
```

## **NOM**

mkfifo - Créer un fichier spécial FIFO

## **SYNOPSIS**

```
#include <sys/types.h>
```

```
#include <sys/stat.h>
```

```
int mkfifo(const char *pathname, mode_t mode);
```

La fonction mkfifo() crée un fichier spécial FIFO (tube nommé) à l'emplacement pathname. mode indique les permissions d'accès (cf. Open)

## **VALEUR RENVOYÉE**

La valeur renvoyée par mkfifo() est 0 si elle réussit, ou -1 si elle échoue, auquel cas errno contient le code d'erreur.

## Les tubes nommés: exemple programme writer

Le programme **writer.c** envoie un message sur le tube *mypipe*

```
int main() {
    int fd;
    char message[100];
    sprintf(message, "bonjour du writer [%d]\n", getpid());
    //Ouverture du tube mypipe en mode écriture
    if(mkfifo("mypipe", S_IRUSR | S_IWUSR)==-1){
        perror("Erreur mkfifo");
        exit(1); // erreur
    }
    fd=open("mypipe",O_WRONLY);
    printf("ici writer[%d] \n", getpid());
    if (fd != -1) { // Dépôt d'un message dans le tube
        write(fd, message, strlen(message) + 1);
    } else
        printf(" désolé, le tube n'est pas disponible \n");
    close(fd);
    return 0;
}
```

# Les tubes nommés: exemple programme reader

Le programme `reader.c` lit un message à partir du tube `mypipe`

```
int main() {
    int fd, n;
    char message[100];
    // ouverture du tube mypipe en mode lecture
    fd = open("mypipe", O_RDONLY);
    printf("ici reader[%d] \n", getpid());
    if (fd != -1) {
        //récupérer un message du tube,
        // taille maximale 100.
        while ((n = read(fd, message, 100)) > 0)
            // n est le nombre de caractères lus
            printf("%s\n", message);
    } else
        printf("désolé, le tube n'est pas disponible\n");
    close(fd);
    return 0;
}
```

## Les tubes nommés: exécution des programmes

- Après avoir compilé séparément les deux programmes, il est possible de lancer leurs exécutions en arrière plan.
- Les processus ainsi créés communiquent via le tube de communication mypipe.
- Lancement de l'exécution d'un **writer** et d'un **reader**:

```
$ ./writer&  reader&
[1] 1156
[2] 1157
ici writer[1156]
ici reader[1157]
bonjour du writer [1156]
[2] Done          reader
[1] + Done        writer
```

## Les tubes nommés: exécution des programmes

- Lancement de l'exécution de **deux writers** et d'**un reader**.

```
$ writer& writer& reader&
[1] 1196
[2] 1197
[3] 1198
ici writer[1196]
ici writer[1197]
ici reader[1198]
bonjour du writer [1196]
bonjour du writer [1197]
[3] Done      reader
[2] + Done    writer
[1] + Done    writer
```

## Les tubes nommés: remarques

- Par défaut, l'ouverture d'un tube nommé est **bloquante** (spécifier `O_NONBLOCK` sinon) :
  - Si un processus ouvre un tube nommé en lecture alors qu'il n'y a aucun processus qui ait fait une ouverture en écriture  
→ le processus sera bloqué jusqu'à ce qu'un processus effectue une ouverture en écriture (resp. lecture).
- Attention aux situations **d'interblocage**