

# La communication IPC

Il est nécessaire que les processus communiquent entre-eux (pour le partage d'information, la répartition des calculs, la modularité et la facilité). La communication inter-process sont très courant sous UNIX et servent à résoudre ce problème.

## Différentes options

- Fichiers, cependant c'est très lent et difficile à synchroniser
- Tube nommé ou non-nommé
- Files de messages
- Mémoire partagée, qui a l'avantage d'être très rapide
- Socket (échanges via le réseau) qui est universel

## Les tubes

Les tubes sont des petits fichiers géré en file circulaire, ils sont si petit qu'ils sont souvent en cache (ce qui est donc très efficace). Si le message devient trop grand, il sera alors découpé en blocs.

### Tubes non-nommés

Les tubes non-nommés sont des tubes temporaires, ils sont alloué via l'appel système `pipe()`

Il existe différents tubes standards :

- `stdin` tube de lecture (via le clavier, genre `scanf`)
- `stdout` tube de sortie (affichage à l'écran, genre `printf`)
- `stderr` est un tube de sortie pour les messages d'erreurs

Il est ainsi possible de rediriger ces tubes.

### Opérations

- Ecriture dans le tube avec appel système `write(int h, char* b, int s)` (h étant le tube, s les premiers octets, et b le buffer)
- Lecture dans le tube avec appel système `read(int h, char* b, int s)`
- Fermeture du tube via `close(int h)`

Note les fonctions `read` et `write` retournent 0 si on tente d'écrire ou de lire un tube sans qu'il n'y a pas de processus à l'autre bout du tube.

## Exemple

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/wait.h>

int main(void) {
    int tube[2];
    char buffer[255];

    /* On crée le tube et on note les identifiant entrée et sortie dans le tableau */
    pipe(tube);

    /* On crée un nouveau processus */
    switch(fork()) {
        case -1:
            printf("Erreur fork()\n");
            exit(-1);

        /* Pour le processus fils */
        /* Le processus fils va lire le processus tube[0] pour avoir la lecture en écriture */
        /* Le buffer va être la variable où les données vont être écrites */
        /* Et enfin 's' est la taille que l'on va récupérer */
        case 0:
            /* Si le tube est vide, read va attendre que le tube soit rempli */
            read(tube[0], buffer, 254);
            printf("Message: %s\n", buffer);
            break;

        /* Pour le processus père : */
        /* Ici on écrit "salut à toi" dans le tube en écriture (tube[1]), le buffer va donc contenir le message */
        /* Le 's' va contenir la longueur du buffer */
        /* Ainsi le message va être envoyé au fils */
        default:
            strncpy(buffer, "salut a toi", 12);
            write(tube[1], buffer, strlen(buffer));
    }
```

```

/* Ici on attends que le processus fils meurt, sinon le read du fils retournera 0 car il n'y aura plus le processus
à l'autre bout car le programme sera terminé */
wait(NULL);
}
return EXIT_SUCCESS;
}

```

## Redirections

Par défaut les 3 tubes standard sont dirigé vers le stdout (ou stderr si configuré autrement).

On peut également rediriger ces tubes, ainis ce qui était affiché à l'écran est alors dirigé automatiquement dans le tube ou peut être lu à partir d'un tube.

Utilisation en shell

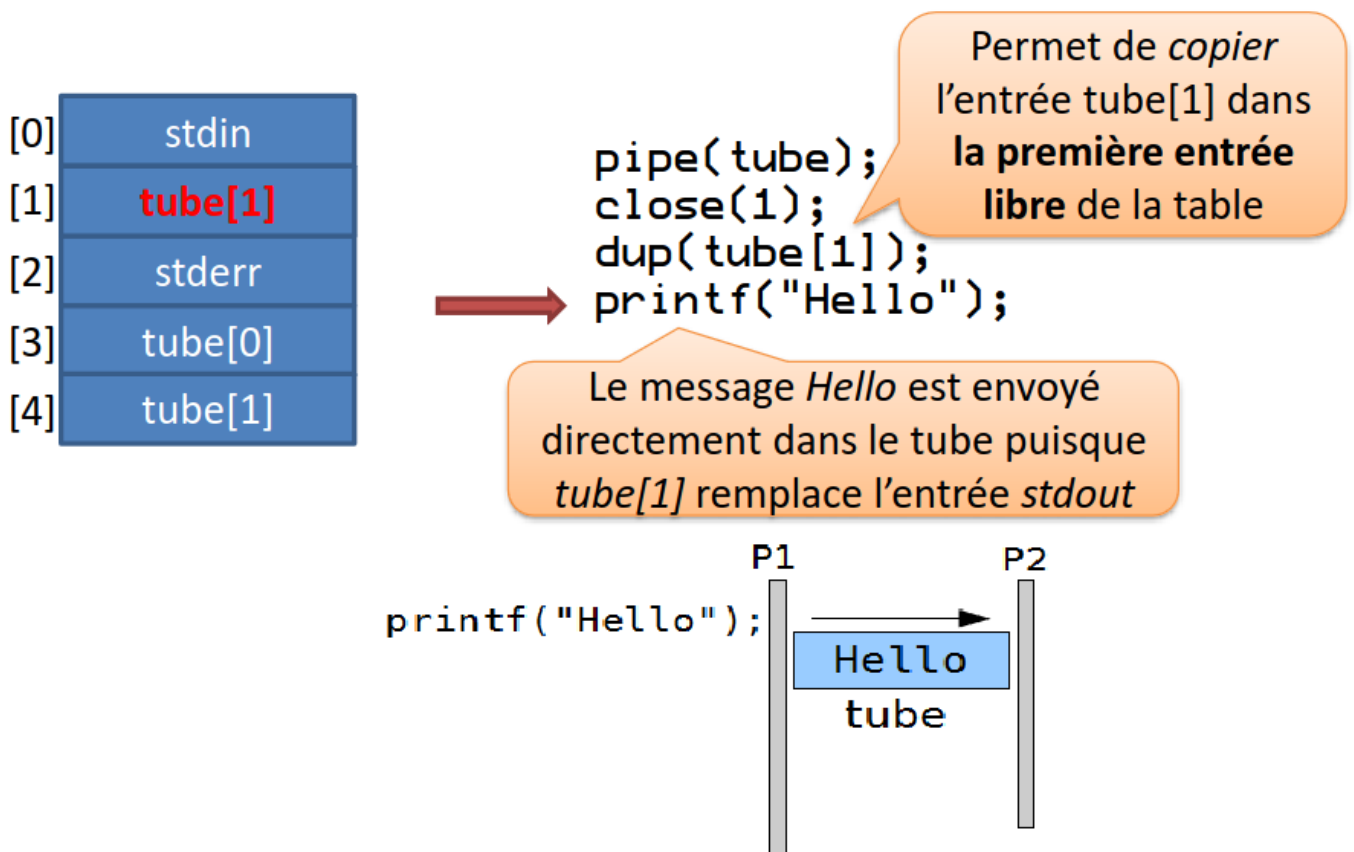
```

# On liste les fichiers et on récupère toutes les lignes contenant "dia"
# grep prends comme entrée le résultat du ls
# C'est le shell qui va automatiquement rediriger le stdout du ls comme le stdin du grep
ls | grep "dia"

```

Fonctionnement

Voici un exemple de redirection :

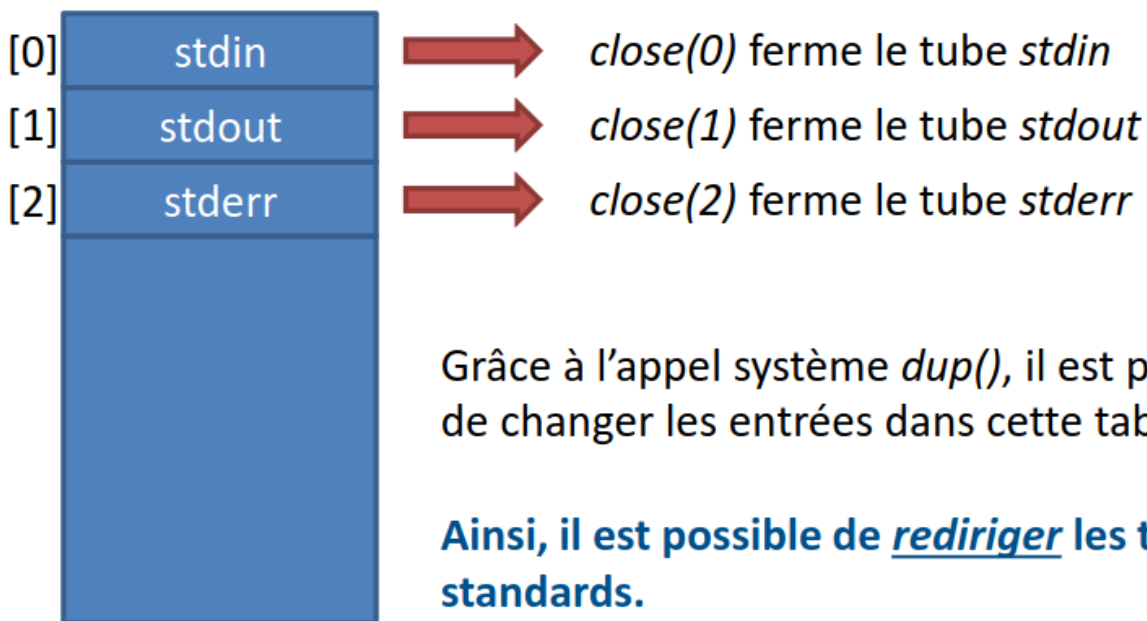


Dans cet exemple :

1. On crée un tube
2. On ferme le stdout
3. On copie notre sortie de tube comme étant le stdout
4. On écrit dans le stdout → donc dans notre tube

## – Fonctionnement

- **Chaque** processus dispose d'une table de descripteurs. Les 3 premières entrées sont :



Exemple en C

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/wait.h>

int main(void) {
    int tube[2];
    char buffer[255];

    /* On crée notre nouveau tube */
    pipe(tube);
```

```

switch(fork()) {
    case -1:
        printf("Erreur fork()\n");
        exit(-1);

    /* Pour le processus fils */
    case 0:
        /* On ferme le stdin */
        close(0);
        /* On copie l'entrée du nouveau tube pour remplacer le stdin */
        dup(tube[0]);
        /* On lit depuis le stdin (on lit donc depuis le tube) */
        scanf("%[^\n]*c", buffer);
        /* On affiche le message stdout */
        printf("Message: %s\n", buffer);
        break;

    /* Pour le processus père */
    default:
        /* On ferme le stdout */
        close(1);
        /* On copie la sortie du tube dans le stdout */
        dup(tube[1]);
        /* On print un message vers le stdout, qui a été redirigé vers le nouveau tube */
        printf("salut a toi\n");
        /* On force le printf a se faire maintenant */
        fflush(stdout);
        /* On attends que le processus fils meurre pour éviter de causer une erreur de lecture du tube */
        wait(NULL);
}
return EXIT_SUCCESS;
}

```

Autre exemple (avec execl)

Lorsque l'on redirige un pipe, le pipe reste redirigé si on exécute un autre programme par après avec `execl`, on peut donc passer l'output d'un programme dans un autre programme. Voici un exemple de pipe qui prends le résultat du `ls` et compte le nombre de lignes, c'est l'équivalent de `ls | wc -l`. Notez cependant que les path de `ls` et `wc` sont **très certainement** différent sur votre système, pour connaitre le PATH réel faites la commande `whereis ls` et `whereis wc`.

```

#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/wait.h>
#include <sys/types.h>

int main(void) {
    int tube[2];

    /* On crée un nouveau tube */
    pipe(tube);

    /* On crée un premier enfant */
    if (fork() == 0) {
        /* On ferme le stdout */
        close(1);
        /* On redirige la sortie du tube dans le stdout */
        dup(tube[1]);

        /* On ferme les tubes pour laisser uniquement le stdin et stdout */
        close(tube[0]);
        close(tube[1]);

        /* On exécute le ls */
        execl("/run/current-system/sw/bin/ls", "/run/current-system/sw/bin/ls", NULL);
        /* Puis ce que rien n'arrive après un execl le reste du code ne s'exécutera pas */
    }

    /* On crée un deuxième enfant */
    if (fork() == 0) {
        /* On ferme le stdin */
        close(0);
        /* On remplace le stdin par le tube[0] */
        dup(tube[0]);
        /* On ferme les tubes pour laisser uniquement les stdin et stdout */
        close(tube[0]);
        close(tube[1]);

        /* On exécute wc -l ça récupère le stdin du ls */
        execl("/run/current-system/sw/bin/wc", "/run/current-system/sw/bin/wc", "-l", NULL);
    }
}

```

```
}

/* On ferme le tube[0] et tube[1] pour laisser uniquement le stdin et stdout */
close(tube[0]);
close(tube[1]);

/* On attends la mort des fils pour mourrir aussi */
wait(NULL);
wait(NULL);
return EXIT_SUCCESS;
}
```

## Tubes nommés

Les tubes nommés sont permanent via des fichiers spéciaux dans le filesystem.

On peut en créer un en utilisant `mkfifo(const char* nom, mode_t mode)` (le nom préise le nom du tube et le mode précise les permissions).

Les processus non-només sont liés entre père et fils, tandis qu'ici les processus nommés peuvent être utilisé par des processus qui bien que sont complètement indépendant l'un de l'autre.

Un processus peut ouvrir un tube en utilisant `open(const char* nom, int flags)` (qui est bloquant par défaut tant que le tube n'est pas ouvert des deux cotés), les flags définissent le mode d'ouverture (écriture, lecture ou les deux bien que cela ne soit pas recommandé).

On peut écrire dans un pipe avec `write(int fd, char* buf, int size)` et lire avec `read(int fd, char* buf, int size)`

On peut enfin fermer un tube avec `close(int fd)`

## Mémoire partagée

La mémoire partagée est un moyen très commun pour partager des informations entre processus, la zone de mémoire est commune à plusieurs processus. La taille est complètement configurable (comme avec malloc) et après un fork, le processus fils hérite de la mémoire partagée.

## Shmget - Allocation

L'allocation se fait via `int shmget(key_t key, int s, int fl)` où

- La clé est l'identifiant de la mémoire partagée
- `s` est la taille en octets
- `fl` est le flag de permission sur la zone

## Petite note sur les permissions

JULIA EVANS  
@bork

# unix permissions

drawings.jvns.ca

There are 3 things you can do to a file

↓  
read write execute

ls -l file.txt shows you permissions  
Here's how to interpret the output:

rw- rw- r-- bork staff  
↑ ↑ ↑  
bork (user) staff (group) ANYONE  
can can can  
read & write read & write read

File permissions are 12 bits

setuid setgid  
↓ ↓  
000 user group all  
sticky rwx rwx rwx

For the r/w/x bits:

1 means "allowed"

0 means "not allowed"

110 in binary is 6

So rw- r-- r--  
= 110 100 100  
= 6 4 4

chmod 644 file.txt  
means change the permissions to:

rw- r-- r--  
simple!

setuid affects executables

\$ls -l /bin/ping

rwS r-x r-x root root  
↑  
this means ping always runs as root

setgid does 3 different unrelated things for executables, directories, and regular files



Les permissions se font via un code tel que 0664 :

- Le premier 0 indique que le nombre est en octal et non pas en décimal. Ainsi 0644 c'est 110 110 100 en binaire, et 777 est 1 100 001 001 en binaire.
- Premier 6 → est le propriétaire signifie que le propriétaire peut lire et écrire(read (1) write (1) execute (0) = 110 = 6)
- Deuxième 6 → est le groupe qui peut lire et écrire également (read (1) write (1) execute (0) = 110 = 6)
- Enfin le 4 → les autres utilisateurs peuvent seulement lire (read (1) write (0) execute (0) = 100 = 4)

## Shmat - Récupération de pointeur

L'appel shmat permet de récupérer un pointeur vers la zone mémoire partagée. Sa signature de méthode est la suivante : char\* shmat(int shmid, char\* addr, int flags) où

- char\* est le pointeur retourné
- int shmid est l'identifiant retourné par shmget



- `char* addr` est l'adresse souhaitée (généralement positionnée à 0 pour laisser le système choisir)
- `int flags` pour les paramètres de restriction (par exemple `SHM_RDONLY` donne un pointeur en lecture seule)

## Shmdt - Détacher la zone

L'appel `shmdt` (qui prends en argument le pointeur) va détacher la zone mémoire sans pour autant la libérer.

## Shmctl - Gérer la zone

L'appel `int shmctl(int shmid, int cmd, struct shmid_ds* ds)` permet de gérer la zone de mémoire.

- `shmid` est le descripteur de la zone retourné par `shmget`
- `cmd` détermine l'opération souhaitée (pour supprimer on utilise `IPC_RMID` mais il existe également `IPC_STAT` pour avoir des informations, `IPC_SET` pour modifier les valeurs associées, etc)
- `ds` contient les données en rapport avec les commandes `STAT` et `SET`

## Exemple

Disons que l'on veut faire 2 programme, 1 premier écrit dans la zone mémoire et le deuxième la lit :

- Premier programme :

```
#include <stdlib.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <string.h>

#define SHM_KEY 2324
#define K 1024

int main(void) {
    int shmid;
    char* ptr;

    /* On alloue une zone de mémoire partagée avec l'identifiant 2324, une taille de 1024 octets, et une
    permission totale pour tout le monde */
    shmid = shmget(SHM_KEY, K, 0777|IPC_CREAT);
```

```

/* Récupère un pointeur vers la zone de mémoire partagée */
ptr = shmat(shmid,NULL,0);

/* On copie une chaîne de caractère dans la mémoire partagée */
strcpy(ptr, "Hello !\n");

/* On détache la zone mémoire (ce qui ne la libère pas mais permet qu'un autre processus l'utilise) */
shmdt(ptr);

/* On ferme le programme */
return EXIT_SUCCESS;
}

```

- Deuxième programme :

```

#include <sys/ipc.h>
#include <sys/shm.h>
#include <sys/types.h>
#include <string.h>
#include <stdlib.h>
#include <sys/msg.h>
#include <stdio.h>

#define SHM_KEY 2324
#define K 1024

int main(void) {
    int shmid;
    char *ptr;

    /* On récupère la zone mémoire avec l'identifiant, la taille et le flag */
    shmid = shmget(SHM_KEY, K, 0777);

    /* Si le shmid retourné est < 0 alors c'est que la zone n'a pas été trouvée */
    if (shmid < 0) {
        printf("Erreur SHM\n");
        exit(-1);
    }
}

```

```

/* On récupère le pointeur de la mémoire partagée */
ptr = shmat(shmid, NULL, 0);

/* On print le contenu de la mémoire partagée */
printf("sa %d", IPC_CREAT);
printf("Contenu : %s\n", ptr);

/* On détache la mémoire du programme */
shmdt(ptr);

/* Le shmctl IPC_RMID va détruire la zone mémoire */
shmctl(shmid, IPC_RMID, NULL);

return EXIT_SUCCESS;
}

```

## Commande ipcs pour lister les mémoires partagées

Si vous souhaitez voir la liste des zones partagées on peut utiliser la commande `ipcs`.

```

[snowcode@snowcode:~]$ gcc mempar.c

[snowcode@snowcode:~]$ ./a.out

[snowcode@snowcode:~]$ ipcs

----- Message Queues -----
key      msqid    owner    perms    used-bytes  messages

----- Shared Memory Segments -----
key      shmid    owner    perms    bytes      nattch     status
0x00000914 4        snowcode 777      1024       0

----- Semaphore Arrays -----
key      semid     owner    perms    nsems

```

Revision #3

Created 24 October 2023 13:54:07 by SnowCode

Updated 2 January 2024 10:22:23 by SnowCode