

INF2610

Noyau d'un système d'exploitation

Chapitre 4 - Communication Interprocessus

Département de génie informatique et génie logiciel

POLYTECHNIQUE
MONTRÉAL

AFILIÉE À
L'UNIVERSITÉ DE MONTRÉAL



Sommaire

- Segments de données partagés
- Tubes de communication
 - Tubes anonymes
 - Tubes nommés
 - Redirection des entrées et sorties standards
- Signaux
- Exercices

Mise en contexte


- Une **application informatique** est, en général, **composée d'un ensemble de processus/threads** qui **s'exécutent en concurrence et s'échangent au besoin des informations**.
- Les systèmes d'exploitation supportent plusieurs **mécanismes de communication** permettant aux processus de communiquer par des :
 - **segments de données partagés**,
 - **échanges de messages** (tubes de communication, sockets, etc.),
 - **signaux**,
 - **etc..**

Segments de données partagés

- Chaque processus **a un espace d'adressage privé** et **accède à la mémoire physique via son espace d'adressage**.
- Cependant, un processus **peut créer des segments de données** et spécifier qui ont **le droit d'y accéder** (segments de données non privés).
- Pour **pouvoir accéder à un segment de données créé**, un processus doit d'abord **l'attacher à son espace d'adressage (mapper)**.
- **Tous les processus qui ont ce segment attaché à leurs espaces d'adressage pourront accéder au segment.**

Segments de données partagés

- Les appels système POSIX pour les segments de données partagés sont regroupés dans la librairie `<sys/mman.h>` (man 7 shm_overview).
 - L'appel système `shm_open` permet de créer ou de retrouver un segment de données. Il retourne un descripteur de fichier ou -1 en cas d'erreur.
 - L'appel système `mmap` permet d'attacher un segment de données à l'espace d'adressage d'un processus. Il retourne l'adresse du début du segment ou -1.
 - L'appel système `munmap` permet de détacher un segment de données de l'espace d'adressage d'un processus.
 - L'appel système `shm_unlink` permet de supprimer le segment de données lorsqu'il sera détaché de tous les espaces d'adressage.

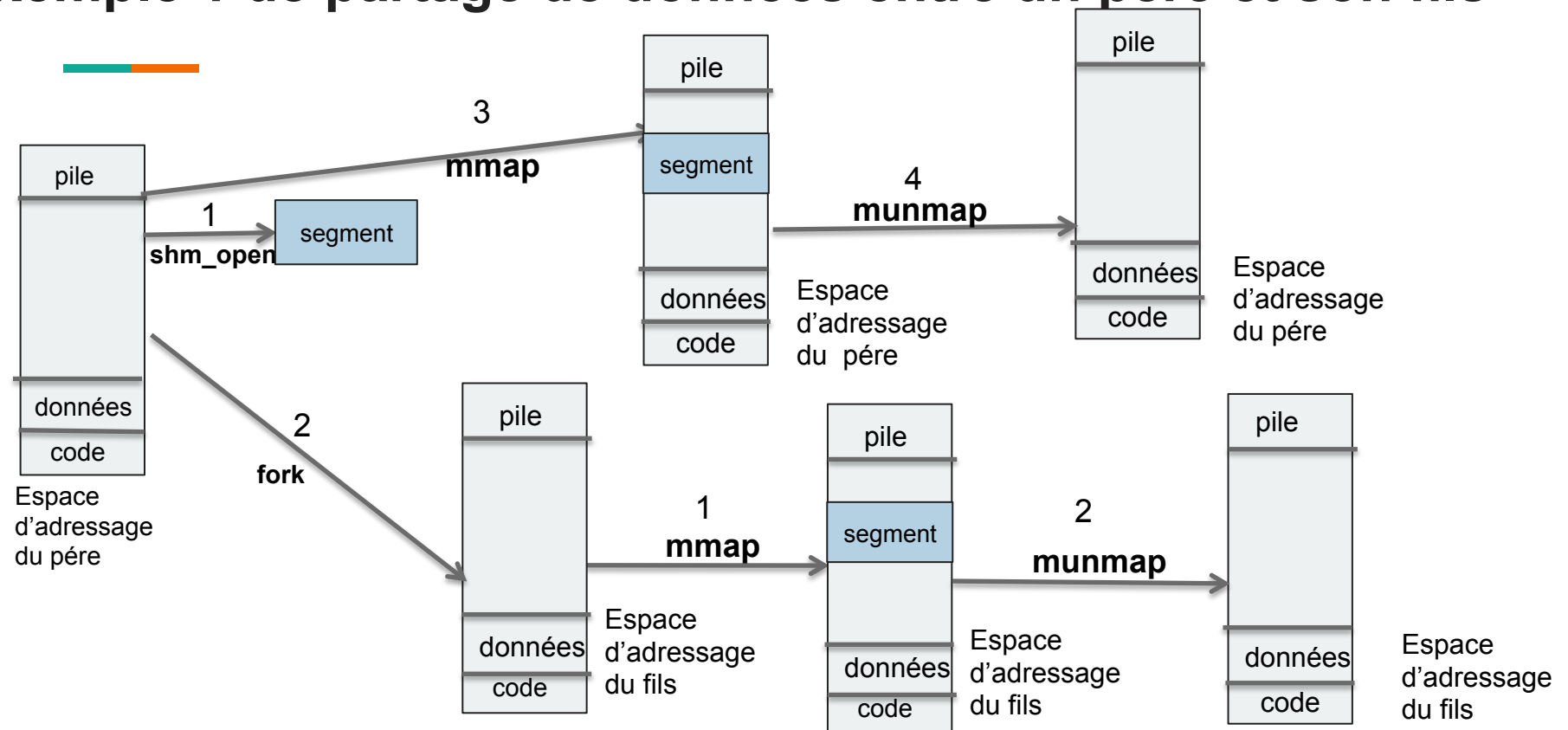


Exemple 1 de partage de données entre un père et son fils

```
./run.sh MemVirt/SegMem1
```

- Le processus principal de ce programme crée, dans l'ordre, un segment de données et un processus fils. Il attache le segment de données créé à son espace d'adressage avant d'écrire dans le segment. Enfin, il détache le segment de son espace d'adressage, attend la fin de son fils, détruit le segment puis se termine.
- Le processus fils attend 2s, attache le segment créé par son père à son espace d'adressage, affiche le contenu du segment, détache le segment de son espace d'adressage puis se termine.

Exemple 1 de partage de données entre un père et son fils




Exemple 1 de partage de données entre un père et son fils

```
....  
int main( ) {  
    off_t  taille = 4096;  
    char  *nom  = "/inf2610shm";  
    int    fd = shm_open(nom, O_RDWR|O_CREAT, 0600 ); // 1  
    if (fd==-1) { exit(1); }  
    if (fork()) { // le père // 2  
        ftruncate(fd, taille);  
        char *ptr = (char *) mmap(NULL, taille, PROT_READ|PROT_WRITE, MAP_SHARED, fd, 0); //3  
        close(fd);  
        if(ptr==NULL) { exit(1); }  
        char msg[100];  
        sprintf(msg, "bonjour du processus %d\n", getpid());  
    }  
}
```


Exemple 1 de partage de données entre un père et son fils

```
    strcpy((char*)ptr,msg);
    printf("Processus %d ecrit dans le segment le message %s\n", getpid(),msg);
    munmap(ptr,taille); // 4
    wait(NULL);
    shm_unlink(nom);
} else { // le fils
    sleep(2);
    char *ptr = (char *) mmap(NULL, taille, PROT_READ|PROT_WRITE, MAP_SHARED, fd, 0);//1
    close(fd);
    printf("Processus %d lit du segment le message %s\n", getpid(), ptr);
    munmap(ptr,taille); //1
}
exit(0); }
```

```
$ ./run.sh MemVirt/SegMem
gcc -o a.out shmp.c -lrt
Processus 12 ecrit dans le segment le message bonjour du processus 12
Processus 13 lit du segment le message bonjour du processus 12
```

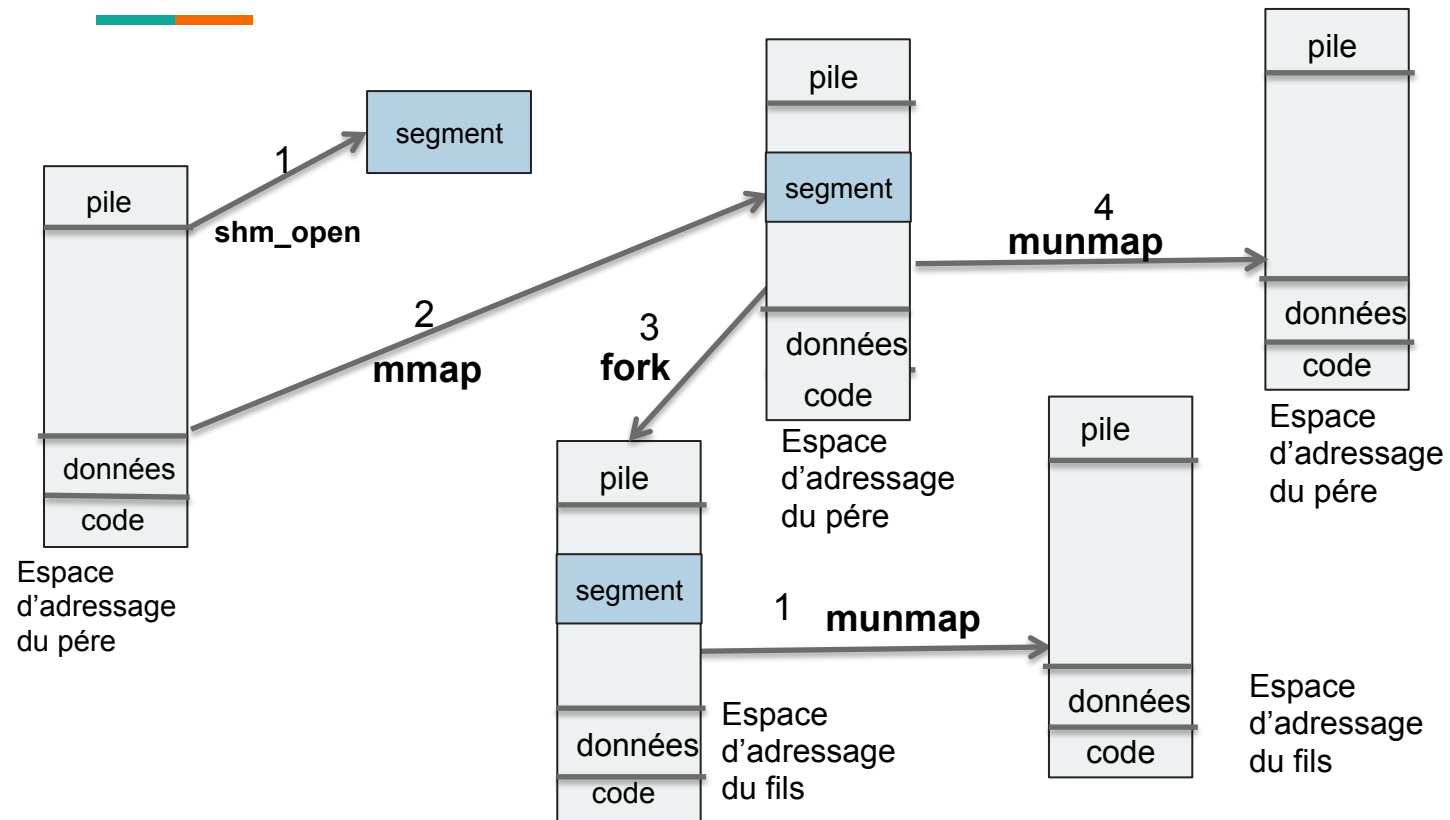


Exemple 2 de partage de données entre un père et son fils

`./run.sh MemVirt/SegMem2`

- Le processus principal de ce programme crée un segment de données, l'attache à son espace d'adressage puis crée un processus fils. Il écrit ensuite dans le segment, avant de le détacher de son espace d'adressage. Enfin, il attend la fin de son fils avant de détruire le segment et se terminer.
- Le processus fils attend 2s et affiche le contenu du segment avant de le détacher de son espace d'adressage et se terminer.

Exemple 2 de partage de données entre un père et son fils



Exemple 2 de partage de données entre un père et son fils

....

```
int main( ) {  
    off_t  taille = 4096;  
    char  *nom  = "/inf2610shm";  
  
    int    fd = shm_open(nom, O_RDWR|O_CREAT, 0600 ); //1  
    if (fd== -1) { exit(1); }  
    ftruncate(fd, taille);  
    char *ptr = (char *) mmap(NULL, taille, PROT_READ|PROT_WRITE, MAP_SHARED, fd, 0); //2  
    if(ptr==NULL) { exit(1);}  
    close(fd);  
}
```

Exemple 2 de partage de données entre un père et son fils

```
if (fork()) { char msg[100];  
    sprintf(msg, "bonjour du processus %d\n", getpid());  
    strcpy((char*)ptr, msg);  
    printf("Processus %d écrit dans le segment le message %s\n", getpid(), msg);  
    munmap(ptr, taille);  
    wait(NULL);  
    shm_unlink(nom);  
} else { // le fils  
    sleep(2);  
    printf("Processus %d lit du segment le message %s\n", getpid(), ptr);  
    munmap(ptr, taille);  
}  
exit(0); }
```

```
$ ./run.sh MemVirt/SegMem2  
gcc -o a.out shmp.c -lrt  
Processus 12 écrit dans le segment le message bonjour du processus 12  
Processus 13 lit du segment le message bonjour du processus 12
```

Exemple 2' de partage de données entre un père et son fils

```
if (fork()) { char msg[100];  
    sprintf(msg, "bonjour du processus %d\n", getpid());  
    sleep(2);  
    strcpy((char*)ptr, msg);  
    printf("Processus %d écrit dans le segment le message %s\n", getpid(), msg);  
    munmap(ptr, taille);  
    wait(NULL);  
    shm_unlink(nom);  
} else { // le fils  
    printf("Processus %d lit du segment le message %s\n", getpid(), ptr);  
    munmap(ptr, taille);  
}  
exit(0); }
```

Accès en
lecture avant
l'écriture

```
$ ./run.sh MemVirt/SegMem2  
gcc -o a.out shmp.c -lrt  
Processus 13 lit du segment le message  
Processus 12 écrit dans le segment le message bonjour du processus 1214
```

Tubes de communication



- Un tube de communication permet une **communication unidirectionnelle** entre des processus (écrivains et lecteurs).
- Il permet de **mémoriser les informations produites** par les processus écrivains. Ces informations sont consommées par les processus lecteurs selon la politique **FIFO**.
- Lorsqu'un lecteur lit une donnée, celle-ci est retirée du tube. **La lecture est donc une opération destructrice**.
- Par défaut, **les lectures et les écritures d'un tube sont bloquantes**.
- Il existe deux types de tubes de communication: **anonymes et nommés**

Tubes anonymes

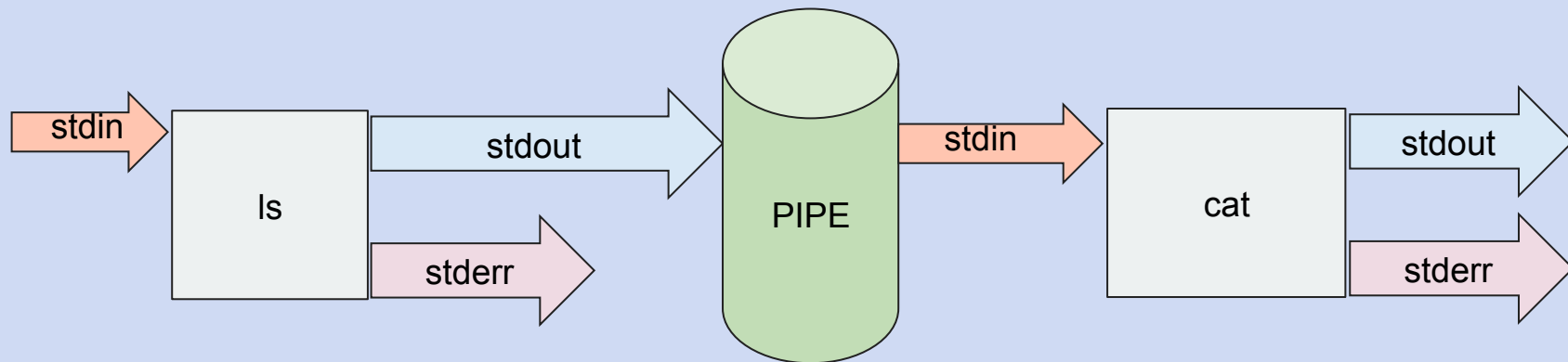


- Un tube anonyme permet d'établir une **communication unidirectionnelle** entre **le créateur du tube et ses descendants**.
- Il est considéré comme un fichier temporaire :
 - Lors de sa création, **deux descripteurs de fichiers lui sont associés** (un pour **les accès en lecture** et l'autre pour **les accès en écriture**).
 - Lorsque tous **les descripteurs de fichiers associés** à ce tube **sont fermés**, **le tube est détruit**.
- Il est possible de **créer un tube anonyme** à l'aide de :
 - **l'opérateur shell “|”** ou encore
 - **l'appel système pipe()** (voir man pipe 2).

Tubes anonymes - opérateur shell « | »

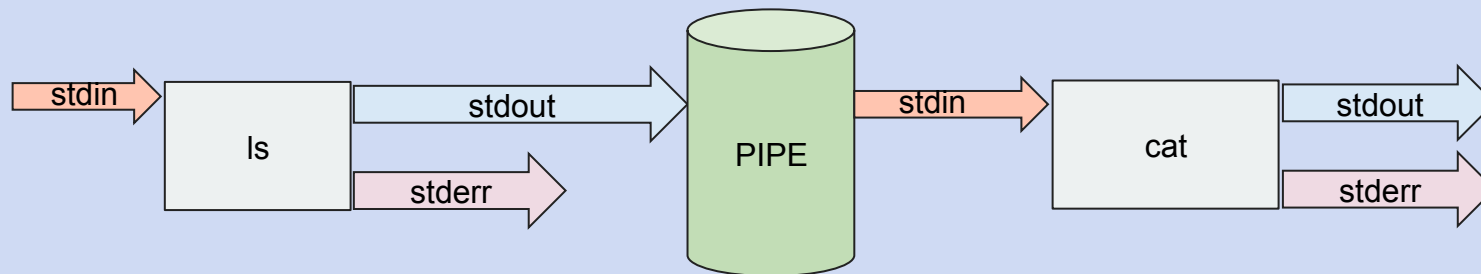
Exemple : `ls | cat`

- `ls` liste les fichiers et les répertoires du répertoire courant alors que `cat` affiche à l'écran les données reçues via son entrée standard.
- L'opérateur « | » redirige, via un tube, la sortie standard du processus exécutant le code `ls` vers l'entrée standard de celui qui exécute le code de `cat`.



Tubes anonymes - opérateur shell « | »

Exemple : `ls | cat`



- Les deux processus s'exécutent en **concurrency**.
- Les **données envoyées par ls**, via sa sortie standard, **sont stockées dans le tube**.
- Le processus écrivain (ls) passe à **l'état bloqué** lorsqu'il essaye d'écrire dans un **tube plein**. Il reste dans cet état tant que le tube est plein.
- Si le **tube est vide** et que le processus lecteur essaye de lire, il **bloquera** tant que le tube est vide et la fin de fichier (EOF) n'est pas atteinte.

Tubes anonymes - appel système pipe()

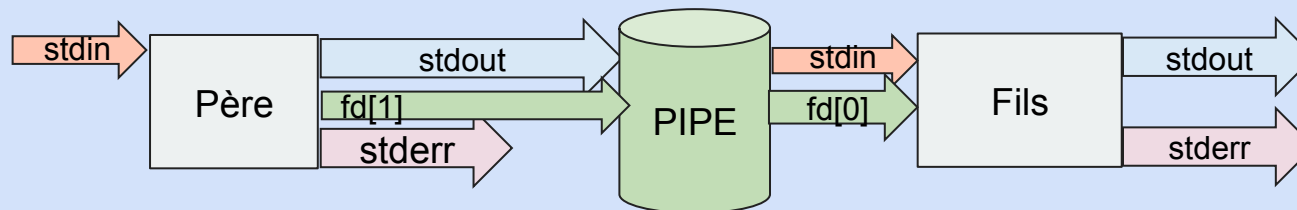
```
int pipe(int fd[2]);
```

- Cet appel crée un **tube de communication anonyme**.
- Une fois l'appel terminé, **le processus appelant a accès à deux descripteurs de fichiers** qui permettent au processus **d'accéder, en lecture et en écriture, au tube créé**.
→ **Ces descripteurs sont ajoutés à la table des descripteurs de fichiers du processus.**
- Les **numéros des descripteurs de fichiers** créés sont récupérés dans le tableau **fd[2]** :
 - **fd[0]** ← le numéro du descripteur à utiliser pour lire du tube : **read(fd[0],);**
 - **fd[1]** ← le numéro du descripteur à utiliser pour écrire dans le tube : **write(fd[1], ...);**
- Cet appel système retourne 0 en cas de succès et -1 en cas d'erreur

Tubes anonymes - Remarques

- Grâce au **clonage** par l'appel fork de la **table des descripteurs de fichiers du père**, les **descendants du père**, créés après la création du tube, **pourront accéder au tube**.
- Si **le tube est vide** et **le nombre d'écrivains est 0** → **une fin de fichier** → **un appel à read retournera 0**.
- Si **le nombre de lecteurs est 0** → **un appel à write générera le signal SIGPIPE** (tentative d'écriture dans un tube rompu). Ce signal est envoyé au processus appelant.
- Lorsque tous les descripteurs de fichiers associés au tube sont fermés (0 lecteurs et 0 écrivains), le tube est détruit.
- Par défaut, **les lectures et les écritures sont bloquantes**.

Comment faire communiquer un père avec un fils ?



1. Père crée un pipe: « `pipe(fd);` ».
2. Père crée un fils: « `fork();` ». Le fils va avoir accès au tube créé car sa table des descripteurs de fichiers est une copie de celle de son père.
3. L'écrivain ferme le descripteur de lecture : « `close(fd[0]);` ».
4. Le lecteur ferme le descripteur d'écriture : « `close(fd[1]);` ».
5. L'écrivain peut maintenant écrire dans le tube via `write(fd[1],....);`
6. Le lecteur peut aussi récupérer des données du tube via `read(fd[0],);`
7. Chaque processus ferme son descripteur de fichier lorsqu'il veut mettre fin à la communication.



Exemple de communication père-fils via un pipe

```
./run.sh Communication/Pipe  
<message>
```

- Cet exemple met en pratique les étapes précédentes pour établir une communication père-fils via un tube anonyme.
- Le programme prend comme argument le message que le processus écrivain va écrire dans le tube. Ce message sera lu par le processus lecteur.

Exemple de communication père-fils via un pipe

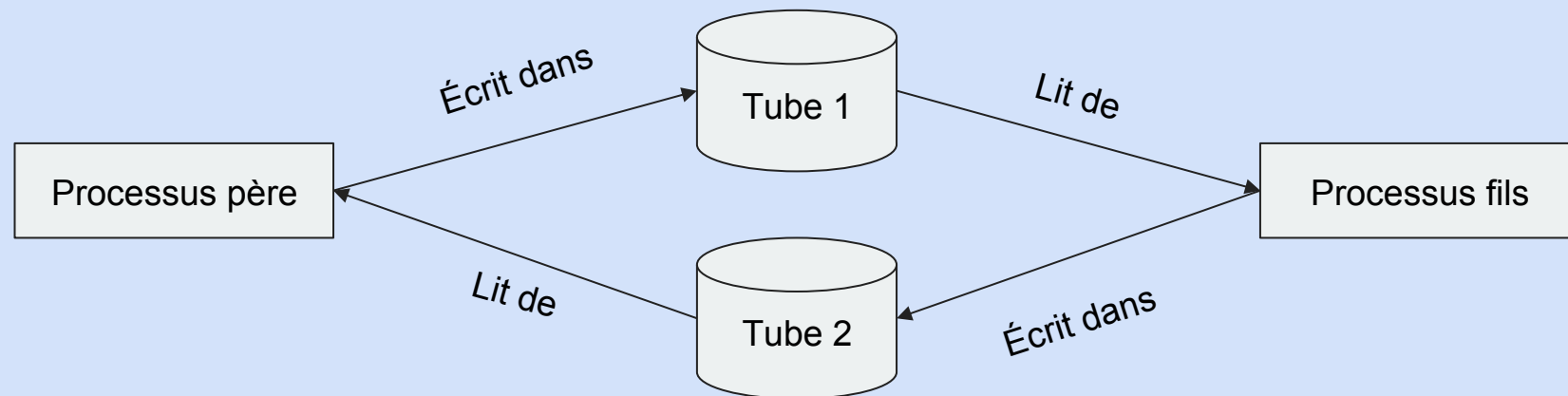
```
....  
int main(int argc, char* argv[]) { ...  
    int fd[2];  
    char buf;  
    pipe(fd);  
    if (fork() == 0) { // le fils est un lecteur  
        close(fd[1]);  
        while (read(fd[0], &buf, 1) > 0) {  
            write(1, &buf, 1);  
        }  
        write(1, "\n", 1);  
        close(fd[0]);  
    }  
}
```

```
else { // le père est un écrivain  
    close(fd[0]);  
    write(fd[1], argv[1], strlen(argv[1])+1);  
    close(fd[1]);  
    wait(NULL); // attend la fin de son fils  
}  
exit(0);  
}
```

Problème d'interblocage si le père ne ferme pas le descripteur d'écriture avant de se mettre en attente de son fils.

Tubes anonymes - Communication bidirectionnelle

- Il est possible d'établir une **communication bidirectionnelle père-fils** en créant deux tubes anonymes (un pour chaque sens de communication).
- Ces deux tubes doivent être créés avant la création du fils.



Tubes nommés



- **Un tube nommé** sert à faire **communiquer des processus d'une même machine** (ayant ou pas un lien de parenté).
- Il **a un nom et existe dans le système de fichiers**. Il est considéré comme un fichier spécial.
- Il **existera dans le système jusqu'à ce qu'il soit détruit explicitement** (unlink).
- Il a une **taille supérieure** à celle des **tubes anonymes**.
- Son **fonctionnement est identique** à celui des **tubes anonymes** (écritures et lectures en FIFO, lectures et écritures bloquantes, fin de fichier, SIGPIPE, etc.).

Tubes nommés – Création et ouverture

- L'appel système `mkfifo` permet de créer un tube nommé :
`int mkfifo(const char* path, mode_t mode)`
 - `path` est un chemin d'accès → **nom et lieu de création du tube.**
 - `mode` est un code de protection → **permissions d'accès au tube.**
 - Retourne 0 en cas de succès et -1 en cas d'erreur.
- Il existe également une commande équivalente: **`mkfifo`**
- Pour accéder à un tube nommé, un processus doit d'abord l'ouvrir
`(open(path,...))` soit en lecture (lecteur) soit en écriture (écrivain).

Tubes nommés – Création et ouverture

- Par défaut, **l'ouverture d'un tube nommé est bloquante** :
 - Un **processus passe à l'état bloqué** s'il essaye d'**ouvrir en écriture un tube nommé sans lecteurs**. Il restera dans cet état tant que le tube n'a aucun lecteur.
 - Un **processus passe à l'état bloqué** s'il essaye d'**ouvrir en lecture un tube nommé sans écrivains**. Il restera dans cet état tant que le tube n'a aucun écrivain.

ATTENTION : Lorsque plusieurs tubes nommés sont utilisés, l'ordre d'ouvertures peut causer un interblocage.

Tubes nommés - Situation d'interblocage

```
// PROCESSUS A  
int f1 = open("tubeA", O_WRONLY);  
int f2 = open("tubeB", O_RDONLY);
```

```
// PROCESSUS B  
int f1 = open("tubeB", O_WRONLY);  
int f2 = open("tubeA", O_RDONLY);
```

- **Processus A attend l'ouverture** par un autre processus **du tube « tubeA » en mode lecture**.
- **Processus B attend l'ouverture** par un autre processus **du tube « tubeB » en mode lecture**.



Exemple fonction mkfifo

`./run.sh Communication/Mkfifo`

Ce programme crée d'abord un tube nommé puis un fils.

Le processus fils ouvre le tube en mode écriture et écrit des messages dans le tube.

Le processus père ouvre le tube en mode lecture puis lit caractère par caractère du tube jusqu'à rencontrer une fin de fichier ou une erreur. Il attend la fin de son fils avant de se terminer.

Tubes nommés - Exemple

```
... int main () {  
    mkfifo("monTube", 0660);  
    if (!fork()) { // fils  
        char m[50];  
        int i;  
        int fd = open("monTube", O_WRONLY);  
        printf("Ici writer [%d]\n", getpid());  
        // Ecriture dans le pipe  
        for(i=0; i<4;i++) {  
            sprintf(m, "mess%d de [%d]\n", i, getpid());  
            write(fd, m, strlen(m) + 1);  
        }  
        close(fd);  
    }  
}
```

```
else { // parent  
    char m;  
    int fd = open("monTube", O_RDONLY);  
    // lire du pipe  
    while (read(fd, &m, 1) > 0) {  
        write(1, &m, 1);  
    }  
    close(fd);  
    wait(NULL);  
}  
exit(0);  
}
```

```
./run.sh Communication/Mkfifo  
gcc -o a.out mkfifo.c  
Ici writer [14]  
mess0 de [14]  
mess1 de [14]  
mess2 de [14]  
mess3 de [14]
```

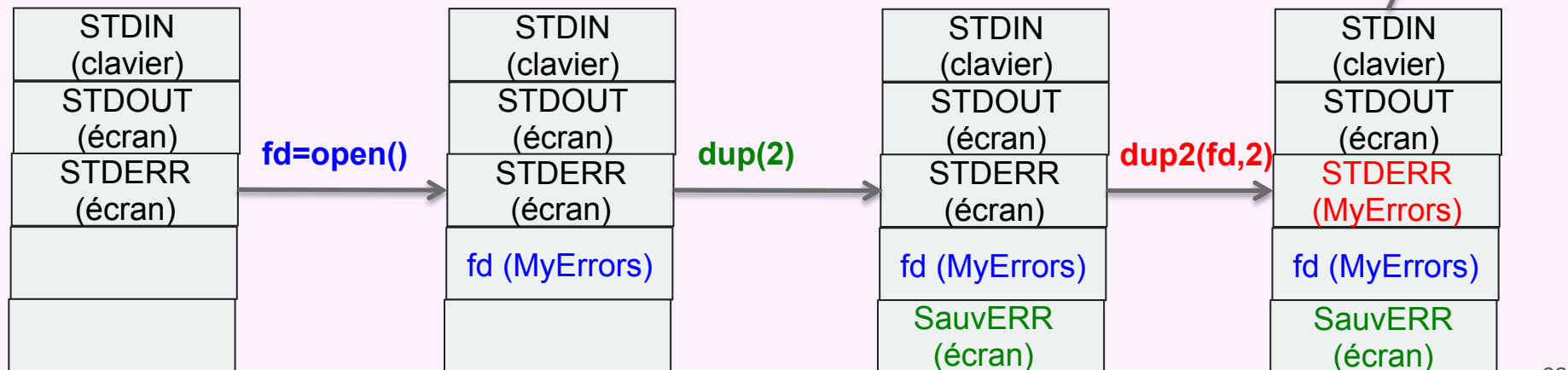
Redirections des entrées et sorties standards

- Par défaut, **l'entrée standard d'un processus** est associée au **clavier**. Les **sorties standard et erreur** sont associées au **moniteur**. Il est cependant **possible de les associer à d'autres fichiers** (ordinaires, tubes, etc.)
- Si vous voulez **recupérer, dans un fichier « MyErrors », les données émises par la sortie erreur d'un processus**, il suffit d'associer **la sortie erreur au fichier**. C'est ce qu'on appelle une redirection de la sortie erreur vers un fichier.
- Les redirections peuvent être réalisées à l'aide des fonctions **dup** et **dup2**.

Redirections des entrées et sorties standards

Exemple :

```
fd = open("MyErrors" , O_WRONLY....);  
SauvERR=dup(2);  
dup2(fd, 2);  
close(fd);
```





Exercice 1

- Donnez un code qui réalise un traitement similaire à celui-ci réalisé par shell lorsqu'on lui sommet la commande **ls | cat**.
- Donnez une autre version du code qui utilise un tube nommé au lieu d'un tube anonyme.

Signaux

- Les systèmes d'exploitation de la famille Unix offrent, au niveau processus, les signaux comme mécanisme de notification d'événements/erreurs et de réactions à ces événements/erreurs.
- Ils gèrent un ensemble fini de signaux. Chaque signal :
 - a un nom, au moins un numéro,
 - a un gestionnaire (handler), et
 - est généralement associé à un événement/erreur.
- La commande `man 7 signal` permet d'afficher des informations sur les signaux gérés par le système.

Signaux (2)

- Si un processus tente d'écrire dans un tube de communication rompu (aucun lecteur), le système d'exploitation envoie le signal SIGPIPE au processus pour l'informer de cette erreur. La réaction par défaut au signal est la terminaison du processus.
- Lorsqu'un processus se termine, il informe son père en lui envoyant le signal SIGCHLD. Par défaut, le processus père ignore ce signal.
- **Questions :**
 - Comment envoyer un signal à un processus ?
 - Quand et comment traite-t-il un signal reçu ?
 - Peut-il définir et associer un traitement à un signal ?
 - Peut-il retarder le traitement d'un signal reçu ?
 - Peut-il se mettre en attente d'un signal ?

Signaux (3)

- Chaque processus a trois tables privées dédiées à la gestion des signaux :
 - Table des gestionnaires des signaux (TGS) qui indique pour chaque signal le traitement associé.
 - Table de bits des signaux en attente (TSA) qui indique les signaux reçus mais non encore traités.
 - Table de bits des signaux à différer (MASK) ➔ masque des signaux du processus.

Comment envoyer un signal à un processus ?

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

- Le bit associé au numsignal dans la table TSA du processus pid est mis à 1.
- Le signal est reçu mais pas encore traité.
- Le délai entre la réception et le traitement d'un signal dépend :
 - de l'état du processus récepteur,
 - du masque (le signal ne sera pas traité tant qu'il est dans le masque des signaux.
 - etc.

Comment traite-t-il un signal?

- Pour traiter un signal reçu, un processus :
 - suspend temporairement son traitement en cours,
 - réalise celui associé au signal (adresse du début du code est dans la TGS),
 - reprend le traitement suspendu ou se termine (si le traitement du signal force sa terminaison).
- Chaque signal a un traitement par défaut mais un processus peut associer un autre traitement (gestionnaire) à un signal, à l'exception des signaux **SIGKILL** et **SIGSTOP** :
 - **`sighandler_t signal(int sig, sighandler_t handler);`**
 - **`int sigaction(int sig, const struct sigaction* action, struct sigaction* oldact);`****Gestionnaire : SIG_DFL, SIG_IGN, ou une fonction du processus.**

➔ Mise à jour de la TGS

Peut-il différer le traitement d'un signal?

- Pour différer le traitement d'un signal, il suffit de l'ajouter dans le masque des signaux du processus :

`int sigprocmask(int how, const sigset_t* set, sigset_t* oldset)`

Dépendamment du paramètre `how`, cette fonction ajoute, retire des signaux du masque ou encore remplace le masque courant par un autre :

- `SIG_BLOCK` : pour **ajouter** les signaux de `set` au masque.
- `SIG_UNBLOCK`: pour **retirer** les signaux de `set` du masque.
- `SIG_SETMASK`: pour **remplacer** le masque courant par `set`.

Les signaux **`SIGKILL`** et **`SIGSTOP`** ne peuvent pas figurer dans le masque.

- L'appel système **`sigpending`** permet de récupérer les signaux en attente (reçus mais non encore traités).

`int sigpending(sigset_t * set).`

Peut-il attendre un signal ?

- L'appel système `pause()` bloque le processus appelant jusqu'à la réception d'un signal :
`int pause()`.
- La fonction `sleep(secs)` bloque le processus appelant jusqu'au prochain signal ou jusqu'à la fin du délai de `secs` secondes :
`unsigned int sleep(unsigned int secs)`.
- La fonction `sigsuspend(set)` remplace temporairement le masque des signaux par `set` puis se met en attente d'un signal :
`int sigsuspend(const sigset_t *sigmask)`.



Exemple envoi d'un signal et gestionnaire personnalisé

***avec signal()**

```
./run.sh Communication/  
Signal
```

Ce programme définit un gestionnaire et l'assigne au signal SIGINT.

Le programme attend ensuite le prochain signal, exécute son gestionnaire, puis se termine.

Signaux - Exemple

....

```
void sigintHandler(int num) {  
    printf("signal SIGINT reçu par %d\n »,getpid());  
}
```

```
int main() {  
    printf("assigner un gestionnaire personnalise a SIGINT pour le processus %d\n", getpid());  
    signal(SIGINT, sigintHandler);  
    printf("c'est fait!\n");  
    printf("appuyez sur CTRL+C!\n");  
    pause();  
    return 0;  
}
```

```
$ ./run.sh Communication/Signal  
gcc -o a.out signal.c  
assigner un gestionnaire personnalise a SIGINT pour le processus 13  
c'est fait!  
appuyez sur CTRL+C!  
^Csignal SIGINT reçu par 13
```



Exercice 2

Modifiez / complétez le programme suivant pour que le père capte le signal SIGCHLD et traite celui-ci en affichant « SIGCHLD reçu par proc. pid », avant de se terminer.

Exercice 2

```
... int main() {  
...  
pid_t cpid=fork();  
if (cpid==0) { // fils  
    pause();  
    exit(0);  
} // Pere  
sleep(5);  
printf("proc. %d envoie SIGUSR1 au proc.%d\n",getpid(),cpid);  
    kill(cpid, SIGUSR1);  
    while(1);  
}
```

Exercice 2 (corrigé)

....

```
void handler(int signum) {  
    printf("SIGCHLD reçu par proc. %d!\n", getpid());  
    exit(0);  
}  
  
int main() {  
    signal(SIGCHLD, handler);  
    pid_t cpid=fork();  
    if (cpid==0) { // fils  
        // signal(SIGCHLD,SIG_DFL);  
        pause();  
        exit(0);  
    }
```

```
$ ./run.sh Communication/Kill  
gcc -o a.out kill.c  
proc. 12 envoie SIGUSR1 au proc. 13  
SIGCHLD reçu par proc. 12!
```

```
// Pere  
sleep(5);  
//envoi du signal SIGUSR1 au fils  
printf("proc. %d envoie SIGUSR1 au proc.%d\n",getpid(),cpid);  
kill(cpid, SIGUSR1);  
while(1);  
}
```

Complétez/modifiez le code pour que le père récupère et affiche l'état de terminaison de son fils.

.... Exercice 2 (corrigé)

```
void handler(int signum) { int status, pid;
printf("SIGCHLD reçu par proc. %d!\n", getpid());
pid =wait(&status);
printf("fin du fils %d: %d, %d, %d\n", pid, WIFSIGNALED(status), WTERMSIG(status),
SIGUSR1);
    exit(0);
}

int main() {
    signal(SIGCHLD, handler);
    pid_t cpid=fork();
    if (cpid==0) { // fils
        // signal(SIGCHLD,SIG_DFL);
        pause();
        exit(0);
    }
    // Pere
    sleep(5);
    //envoi du signal SIGUSR1 au fils
    printf("proc. %d envoie SIGUSR1 au proc.%d\n",getpid(),cpid);
    kill(cpid, SIGUSR1);
    while(1);
}
```

Lecture suggérée

- **Chapitre 3 : Communication Interprocessus**

Introduction aux systèmes d'exploitation - Cours et exercices en GNU/Linux, Hanifa Boucheneb & Juan-Manuel Torres-Moreno, 216 pages, édition ellipses, 2019, ISBN : 9782340029651.

- **Capsules de Vittorio** : dup2 et exec perror et errno sprintf
https://www.youtube.com/channel/UCffP7k2AXCttKadZcHsqsYg?view_as=subscriber