

Programmation Système et Réseau



Par : Anis ZOUAOUI

Anis.ZOUAOUI@esprit.ens.tn

Anis.zwawi@gmail.com

Programmation Système et Réseau

Chapitre 3

Communication Inter Processus

Par : Anis ZOUAOUI



PLAN

- Introduction
- Les tubes de communication anonymes
- Les signaux

Introduction

- ✓ Les systèmes d'exploitation actuels offrent la possibilité de créer **plusieurs processus**(threads) concurrents qui **coopèrent** pour réaliser des **traitements complexes** d'un **même** application.
- ✓ Ces processus s'exécutent sur même ordinateur (monoprocasseur) ou multiprocasseur ou sur des différents ordinateurs, et peuvent **s'échanger** des informations(communication inter processus).

Introduction

- ✓ Il existe plusieurs mécanismes de communication interprocessus :
 - Les données communes (variables, fichiers, segments de données)
 - Les signaux
 - Les messages.

Introduction

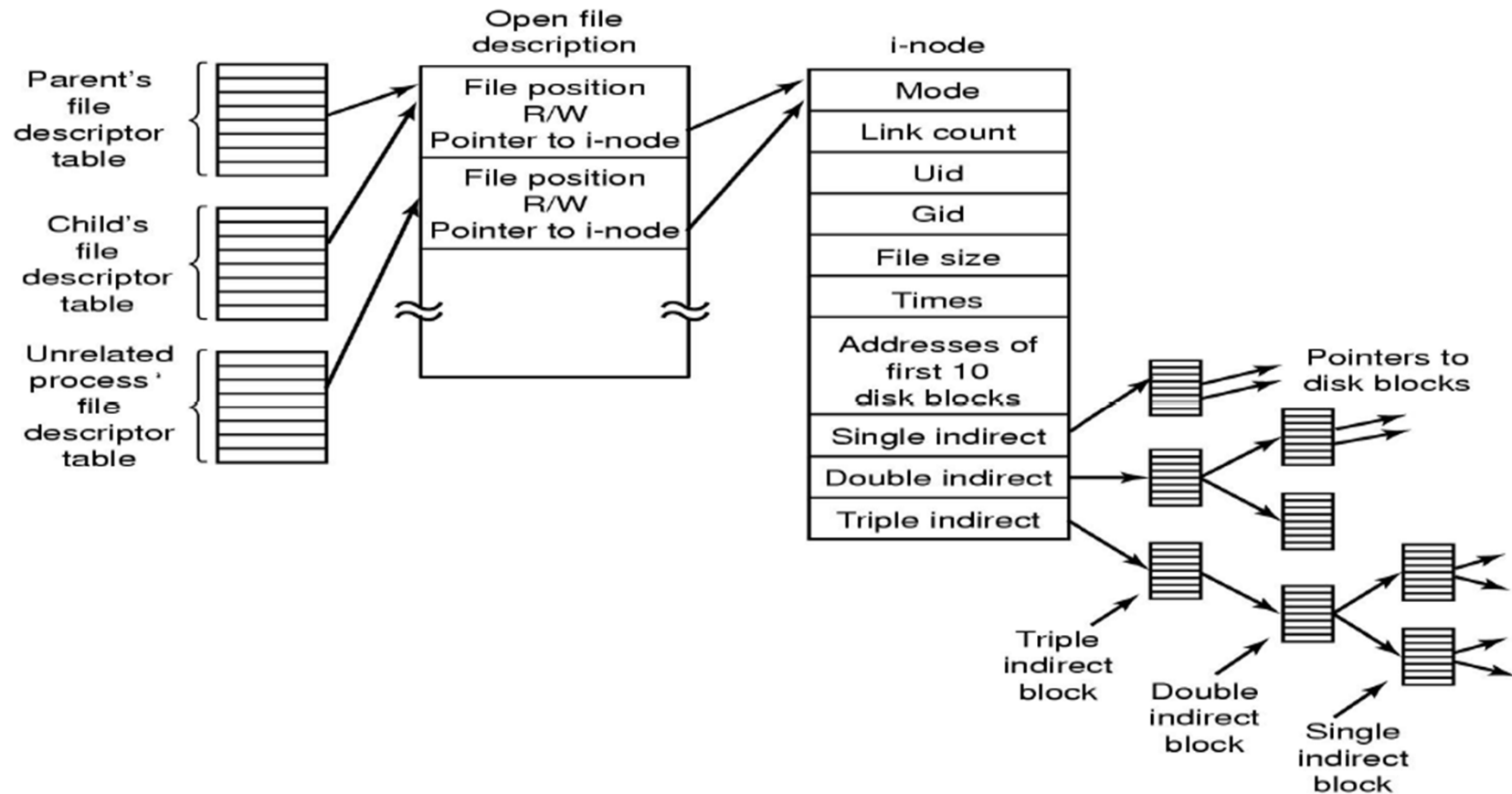
Les threads (POSIX) d'un processus partagent :

- La zone de données globales
- Le code
- La table des descripteurs de fichiers du processus.

Lors de la création d'un processus (fork) :

- la table des descripteurs de fichiers est dupliquée
- Les processus créateur et créé partageront le même pointeur de fichier pour chaque fichier déjà ouvert lors de la création.

Comment ça marche?



Les tubes de communications unix

Définition:

Les tubes (pipes) de communications permettent entre deux ou plusieurs processus s'exécutant sur une même machine d'échanger des informations.

Deux types :

- ❖ Les tubes anonymes
- ❖ Les tubes nommés, qui ont une existence dans le système de fichiers : Donc un chemin d'accès

Les tubes Anonymes : Création

Un tube de communication est créé par l'appel système :

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

Cette appel système crée deux descripteurs de fichiers, il retourne, dans p, les descripteurs de fichiers créés.

- p[0] : contient le descripteur réservés aux lectures à partir du tube.
- p[1] : contient le descripteur réservés aux écritures dans le tube.

Les tubes Anonymes : Création

Les descripteurs créés sont ajoutés aux tables de descripteurs de fichiers du processus appelant.

Seul le processus créateur du tube et ses descendants peuvent accéder au tube. (Duplication de la table de descripteurs de fichiers).

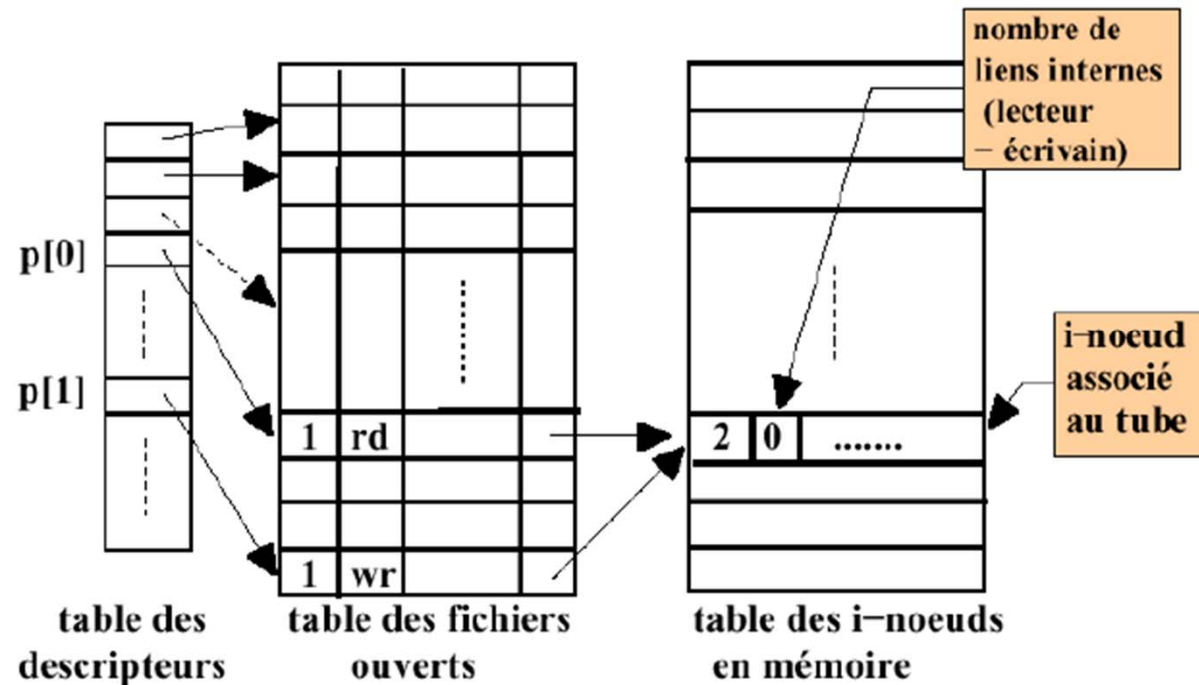
Si le tube n'est pas créé: manque d'espace, l'appel système renvoie -1, sinon il retourne la valeur 0.

L'accès au tubes se fait via les descripteurs de fichiers.

Les tubes Anonymes : Création

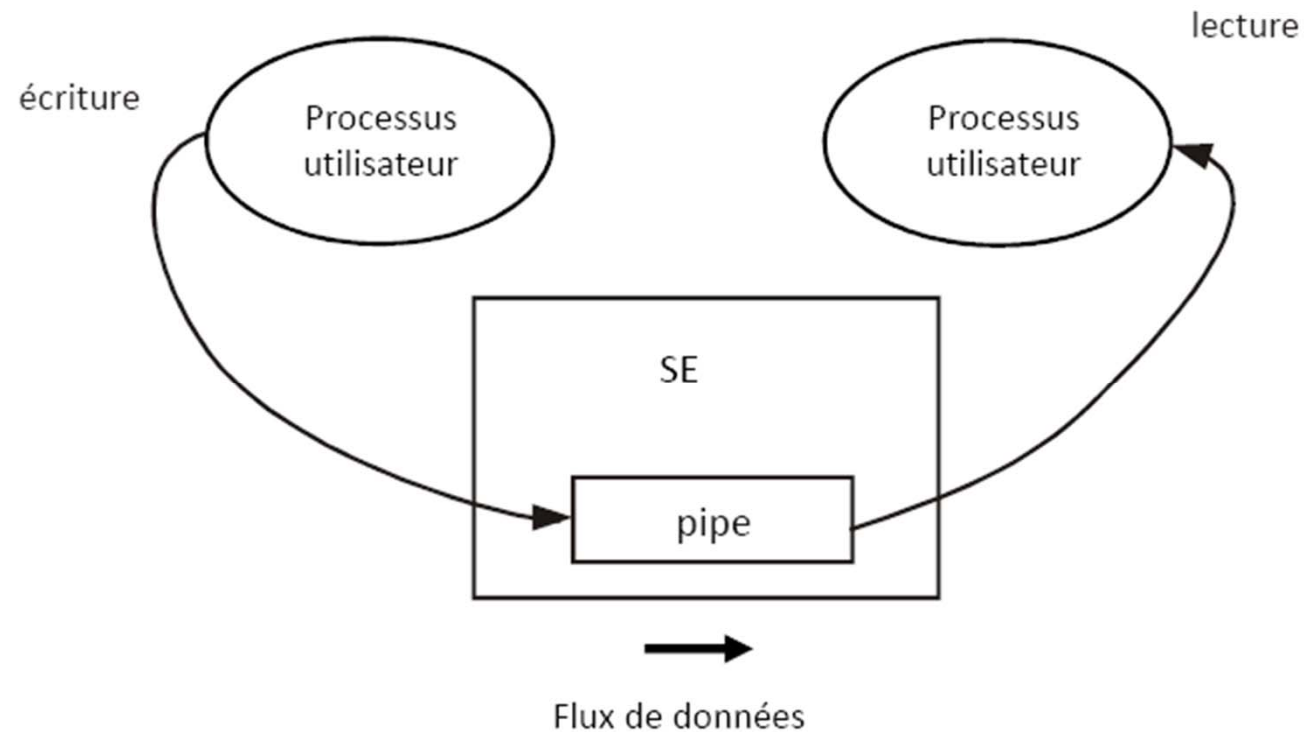
Communications unidirectionnels:

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



Les tubes Anonymes

Communications unidirectionnels:



Les tubes Anonymes

Communications unidirectionnels:

Les tubes anonymes sont, en général, utilisés entre un processus père et son fils.

Scénario:

Père : Crée un tube.

Père : Crée un ou plusieurs fils

Père → Lecteur

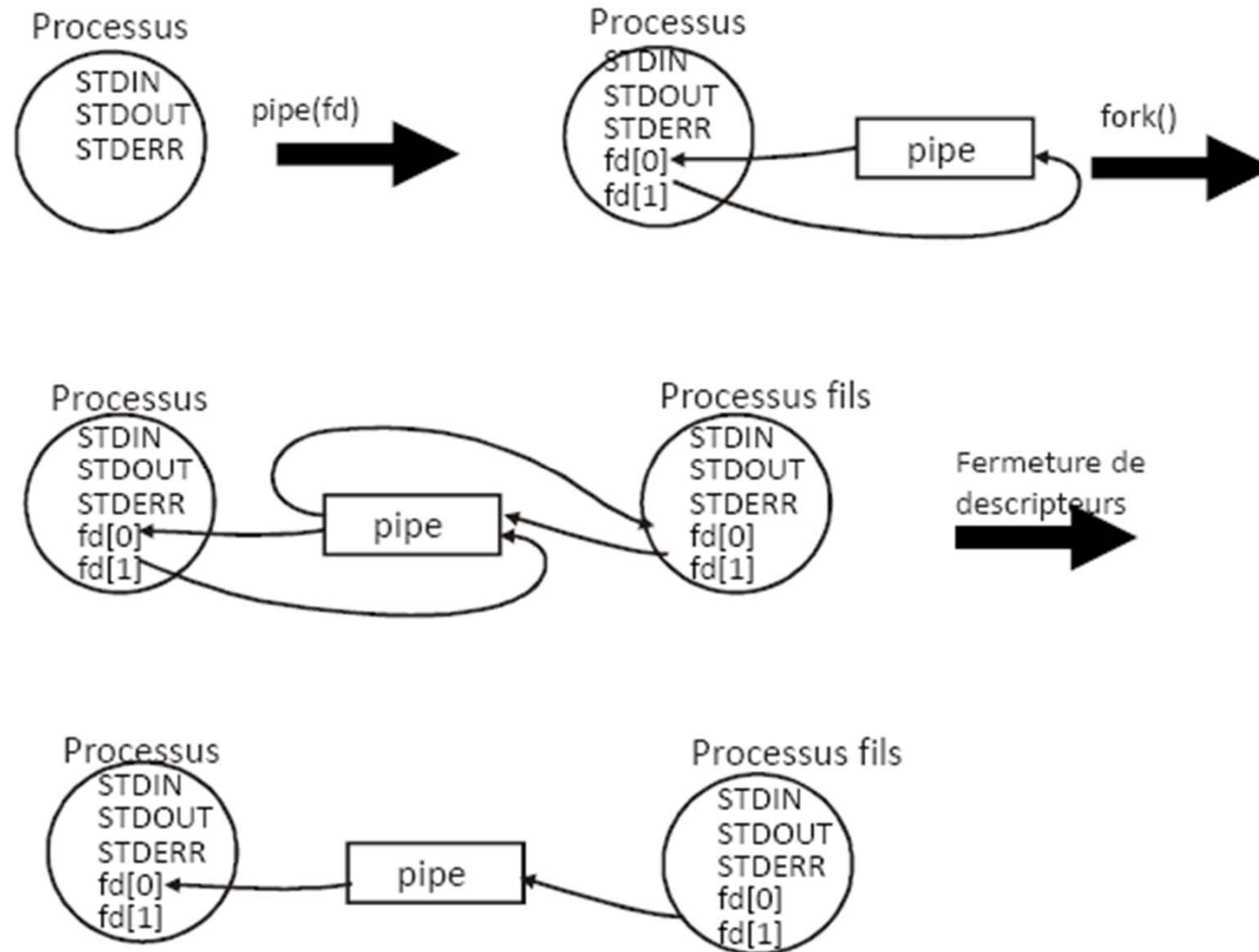
Fils → Ecrivain

Ecrivain : ferme le fichier non utilisé : de lecture.

Lecteur : Ferme le fichier non utilisé : d'écriture

Les processus (lecteur et écrivain) communiquent en utilisant les primitives read et write.

Les tubes Anonymes



Les tubes Anonymes

Communications unidirectionnels:

Exemple 1:

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

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

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

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

```
#include<string.h>
```

```
#include<stdlib.h>
```

```
#define R 0
```

```
#define W 1
```

```
int main()
```

```
{
```

```
    int pid;
```

```
    int fd[2];
```

```
    char message[100];
```

```
    int Nboctets;
```

```
    char* phrase="VOICI MON MESSAGE POUR TOI PERE";
```

```
    system("clear");
```



Example :

```
printf("_____ \n");
printf("\t\tProcessus Courant:: %d\n",(int)getpid());
printf("_____ \n");
```

```
if (pipe(fd) == -1)
{
    perror ("Creation du pipe a échoué ");
    exit(1);
}
```

```
pid=fork();  
if(!pid)  
{  
close(fd[R]); //fermeture du fichier de lecture par le fils  
printf("\t\t\t\t\t\t\t+++++++ \e[m\n");  
printf("\t\t\t\t\t\t\t \e[29m Processus FILS:: %d\n",(int)getpid());  
printf("\t\t\t\t\t\t\t+++++++\n");  
printf("\t\t\t\t\t\t\t \e[29mProcessus FILS::PERE JE T'ENVOIE UN MESSAGE\n");
```

Les tubes Anonymes

Communications unidirectionnels:

Exemple :

```
if (write(fd[W],phrase,strlen(phrase)+1)== -1)
{
    perror("write : Ecriture dans le pipe à échoué ");
    exit(4);
}

close(fd[W]); //fermeture du fichier d'écriture par le fils
//sleep(2);
exit(3);
}

printf("%c[%d;%dm J'attends la terminaison du fils%c[%dm\n",27,1,33,27,0);
wait(NULL);

close(fd[W]); //fermeture du fichier de lecture par le pere
```

Les tubes Anonymes

Communications unidirectionnels:

Exemple :

```
if (( Nboctets = read (fd[0],message, 100)) == -1)
{
    perror ("read : Lecture échoué ");
    exit(5);
}
message[Nboctets]='\0';
printf("%c[%d;%dmMESSAGE RECU :: nbctets = %d Message=
%s%c[%dm\n",27,1,33,Nboctets,message,27,0);
close(fd[R]); //fermeture du fichier d'écriture par le pere
return 0;

}
```

Les tubes Anonymes

Communications unidirectionnels:

Exemple 2:

```
#include<sys/types.h>
#include<sys/wait.h>
#include<unistd.h>
#include<stdio.h>
#include<string.h>
#include<stdlib.h>
#include<assert.h>
#define R 0
#define W 1
int main(int argc,char* argv[])
{
    int tube[2];
    int pid;
    char buf;
    assert(argc==2);
```

```
    if(pipe(tube)==-1)
    {
        perror("Cr ation pipe : ");
        exit(EXIT_FAILURE);
    }

    system("clear");

    printf("_____\\n");
    printf("\\t\\tProcessus Courant:: %d\\n",(int)getpid());
    printf("_____\\n");
```

Les tubes Anonymes

Communications unidirectionnels:

Exemple 2:

```
pid=fork();

if(pid==-1)
{
    perror("Fork echec");
    exit(EXIT_FAILURE);
}

if(!pid)//FILS
{
    close(tube[W]);
    while(read(tube[R],&buf,1) > 0)
    {
        sleep(1);
        write(STDOUT_FILENO,&buf,1);
    }
    write(STDOUT_FILENO,"\n",1);
    close(tube[R]);
    exit(EXIT_SUCCESS);
}
```

```
//PERE
close(tube[R]);
write(tube[W],argv[1],strlen(argv[1])+1);
close(tube[W]);
printf("%c[%d;%dm J'attends la terminaison du\n",27,1,33,27,0);
wait(NULL);
printf("%c[%d;%dm BYE %c[%d;%dm\n",27,1,33,27,0);

return 0;
}
```

Les tubes Anonymes

Communications unidirectionnels:

Redirection des entrées standards et des sorties standards

La duplication de descripteur permet à un processus de créer un nouveau descripteur (dans la table de descripteur) synonyme d'un descripteur déjà existant.

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

```
int dup(int desc);
```

```
int dup2(int desc1, int desc2);
```

dup crée et retourne un descripteur synonyme à desc, le numéro associé au descripteur crée est le plus petit descripteur disponible dans la table des descripteurs des fichiers du processus

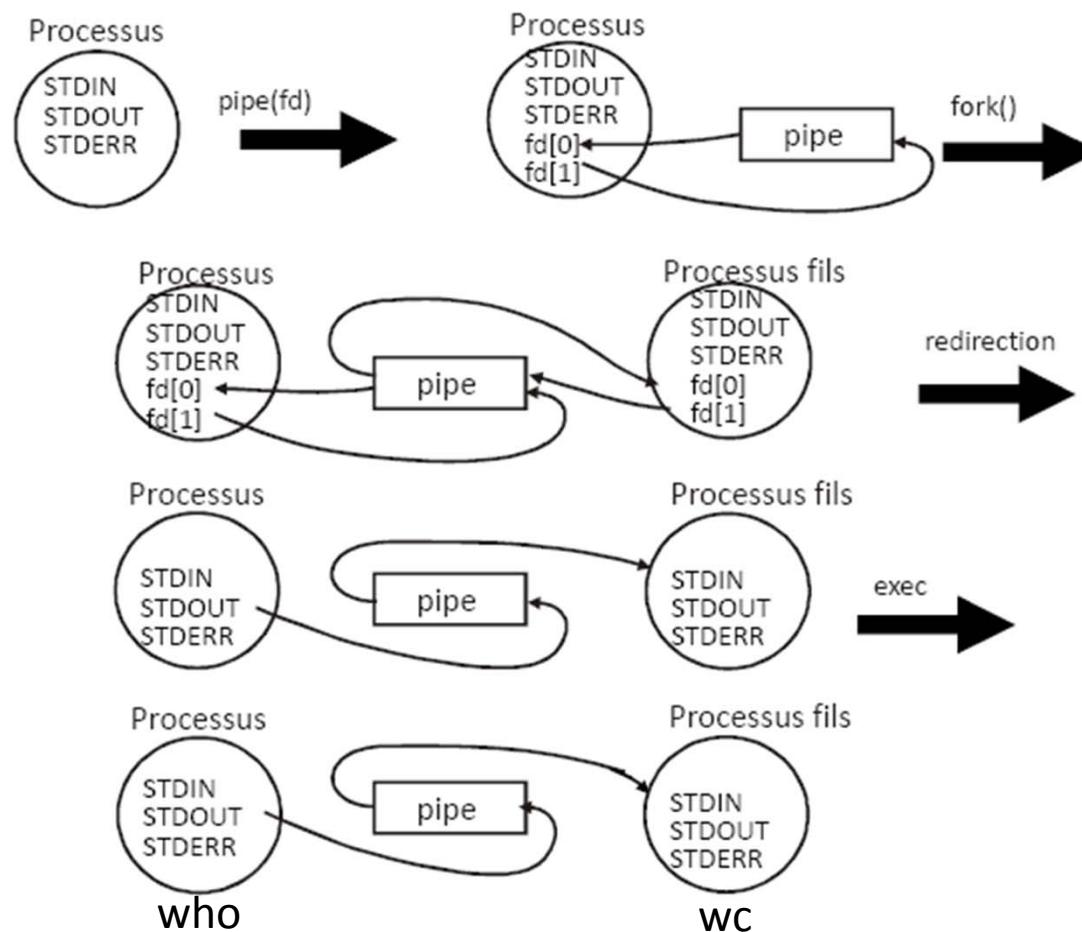
dup2 transforme le desc2 à un descripteur synonyme au descripteur desc1

Les tubes Anonymes

Communications unidirectionnels:

Redirection des entrées standards et des sorties standards

Les fonctions `dup` et `dup2` permettent la redirection des entrées sorties vers les tubes de communications.



Les tubes Anonymes

Communications unidirectionnels:

Exemple 3:

```
int execl (const char *app, const char *arg, ...);
```

app : chemin complet de l'application

arg : paramètre sous forme d'une liste d'argument, termine par pointeur NULL

```
#include<sys/types.h>
#include<sys/wait.h>
#include<unistd.h>
#include<stdio.h>
#include<assert.h>
#include<string.h>
#include<stdlib.h>
#define R 0
#define W 1
```

```
int main(int argc, char* argv[])
{
    int pid;
    int fd[2];
    assert(argc==3);
    system("clear");
    pid=fork();
    if(!pid) {
        close(fd[W]);
        dup2(fd[R],0);
        close(fd[R]);
        execlp(argv[2],argv[2],NULL);
    }
```

Les tubes Anonymes

Communications unidirectionnels:

Exemple 3:

```
else
{
close(fd[R]);
dup2(fd[W],1);
close(fd[W]);
if(execlp(argv[1],argv[1],NULL)==-1)
perror("execlp");

}

return 0;
}
```

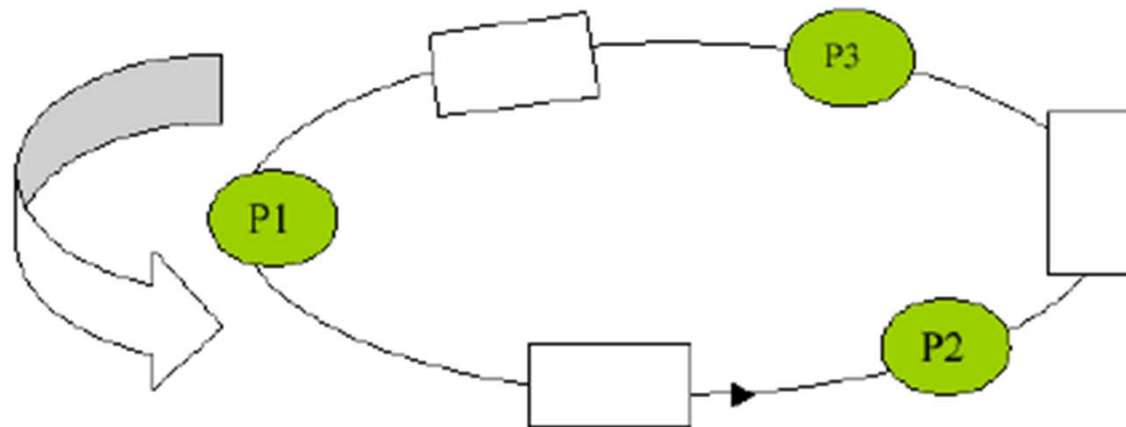
Execlp: la recherche de commande se fait dans la variable d'environnement PATH

Les tubes Anonymes

Communications unidirectionnels:

EXERCICE

On veut établir, en utilisant les tubes nommés, une communication de type anneau unidirectionnel entre trois processus fils. Pour ce faire, la sortie standard de l'un doit être redirigée vers l'entrée standard d'un autre.



Les tubes Anonymes

Communications unidirectionnels:

EXERCICE

```
int main ()
{
    /*1*/

    if (fork()) // création du premier processus
    {
        if(fork())
        {
            /*2*/
            if(fork())
            {
                /*3*/

                while (wait(NULL)>0);

                /*4*/
            } else
            {
                // processus P3
                /*5*/
            }
        }
    }
}
```

Les tubes Anonymes

Communications unidirectionnels:

EXERCICE

```
        execlp("program3", "program3", NULL);

        /*6*/
    }

} else
{    // processus P2

    /*7*/

    execlp("program2", "program2", NULL);

    /*8*/
}
} else
```

Les tubes Anonymes

Communications unidirectionnels:

EXERCICE

```
{ //processus P1

    /*9*/

    execlp("program1","program1", NULL);

    /*10*/
}
/*11*/
}
```

Les signaux

Un signal est une interruption logicielle asynchrone qui a pour but d'informer l'arrivée d'un événement.

Analogies dans la vie courante :

- ▲ Le réveil sonne.
- ▲ Le téléphone sonne.
- ▲ Quelqu'un frappe à la porte.
- ▲ Une fenêtre arrive à l'écran pour annoncer qu'on a du courrier.
- ▲ Un automobiliste fait un appel de phares...
- Un signal peut changer le déroulement d'un processus.
- Les signaux sont dits *asynchrones* : reçus à n'importe quel moment.
- Les signaux sont codés par des entiers et identifiés par des noms.

Les signaux

Le Système d'exploitation gère un ensemble de signaux.

Un Signal a un nom, un numéro, un gestionnaire(handler) et est associé à un type d'événement.

\$man 7 signal : visualiser la liste des signaux et les événements associés.

Les signaux

Le Système d'exploitation associe à chaque signal un traitement par défaut (gestionnaire par défaut):

- ✓ abort (génération d'un fichier core et arrêt du processus);
- ✓ exit(Terminaison du processus sans génération d'un fichier core)
- ✓ ignore(Le signal est ignoré);
- ✓ stop(suspension du processus)
- ✓ Continue(reprendre l'exécution si le processus est suspendu sinon le signal est ignoré).

Par exemple : SIGUSR1 et SIGUSR2 tuent le processus, alors SIGCHLD est ignoré).

Les signaux : Gestionnaire(1)

Lorsqu'un signal est envoyé à un processus, le système **interrompt** l'exécution du processus pour lui permettre de **réagir** au signal → **Exécuter le handler** du signal.

Le système permet à un processus de **redéfinir**, pour certains signaux, leurs gestionnaires. Un processus peut donc indiquer au système **sa réaction** à la réception d'un signal :

- Ignorer le signal (certains ne peuvent pas être ignorés)
- Le prendre en compte
- Exécuter le traitement par défaut
- Le bloquer(le différer)

Les signaux : Gestionnaire(2)

Si un processus choisit de prendre en compte un signal qui lui est destiné (capture d'un signal), il doit spécifier la procédure de gestion du signal.

Exp : la touche d'interruption Ctrl+C génère un signal. Par défaut ce signal arrête le processus. Le processus peut spécifier à ce signal un autre gestionnaire de signal.

Les signaux

Signal	Action par défaut	Événement déclenchant
SIGABRT	A	Généré par <code>abort()</code> .
SIGALRM	T	Expiration d'une alarme.
SIGCHLD	I	Fils terminé ou stoppé (ou continué [XSI]).
SIGCONT	C	Continue l'exécution, si stoppé.
SIGFPE	A	Opération arithmétique erronée.
SIGHUP	T	Hangup, fin de session.
SIGINT	T	Interruption au terminal (C-c).
SIGKILL	T	Termine le processus (*).
SIGPIPE	T	Écriture sur tube sans lecteurs.

(*) ne peut être ni capté ni ignoré

Δ T : termine le processus normalement.

Δ A : termine le processus anormalement (eg, avec fichier core).

Δ I : ignoré.

Δ C : continue un processus stoppé.

Les signaux

Signal	Action par défaut	Événement déclenchant
SIGQUIT	A	Quit au terminal (C-\\).
SIGSEGV	A	Référence mémoire invalide.
SIGSTOP	S	Stoppe l'exécution (*).
SIGTERM	T	Termine l'exécution.
SIGTSTP	S	Envoi Stop au terminal (C-Z).
SIGTTIN	S	Processus en background essayant de lire.
SIGTTOU	S	Processus en background essayant d'écrire.
SIGUSR1	T	À la disposition du programmeur.
SIGUSR2	T	À la disposition du programmeur.

(*) ne peut être ni capté ni ignoré

Δ T : termine le processus normalement.

Δ A : termine le processus anormalement (eg, avec fichier core).

Δ I : ignoré.

Δ C : continue un processus stoppé.

Les signaux : Envoi d'un signal

```
#include<sys/types.h>
#include<signal.h>
int kill(pid_t pid, int sig)
```

Si $\text{pid} > 0$, le signal sig est envoyé au processus pid

Si $\text{pid} = 0$, le signal est envoyé à tous les processus du groupe émetteur.

`kill` retourne 0 en cas de succès et -1 en cas d'erreur.

Les signaux : Réception d'un signal

Le système vérifie si un processus a reçu un signal aux transitions suivantes :

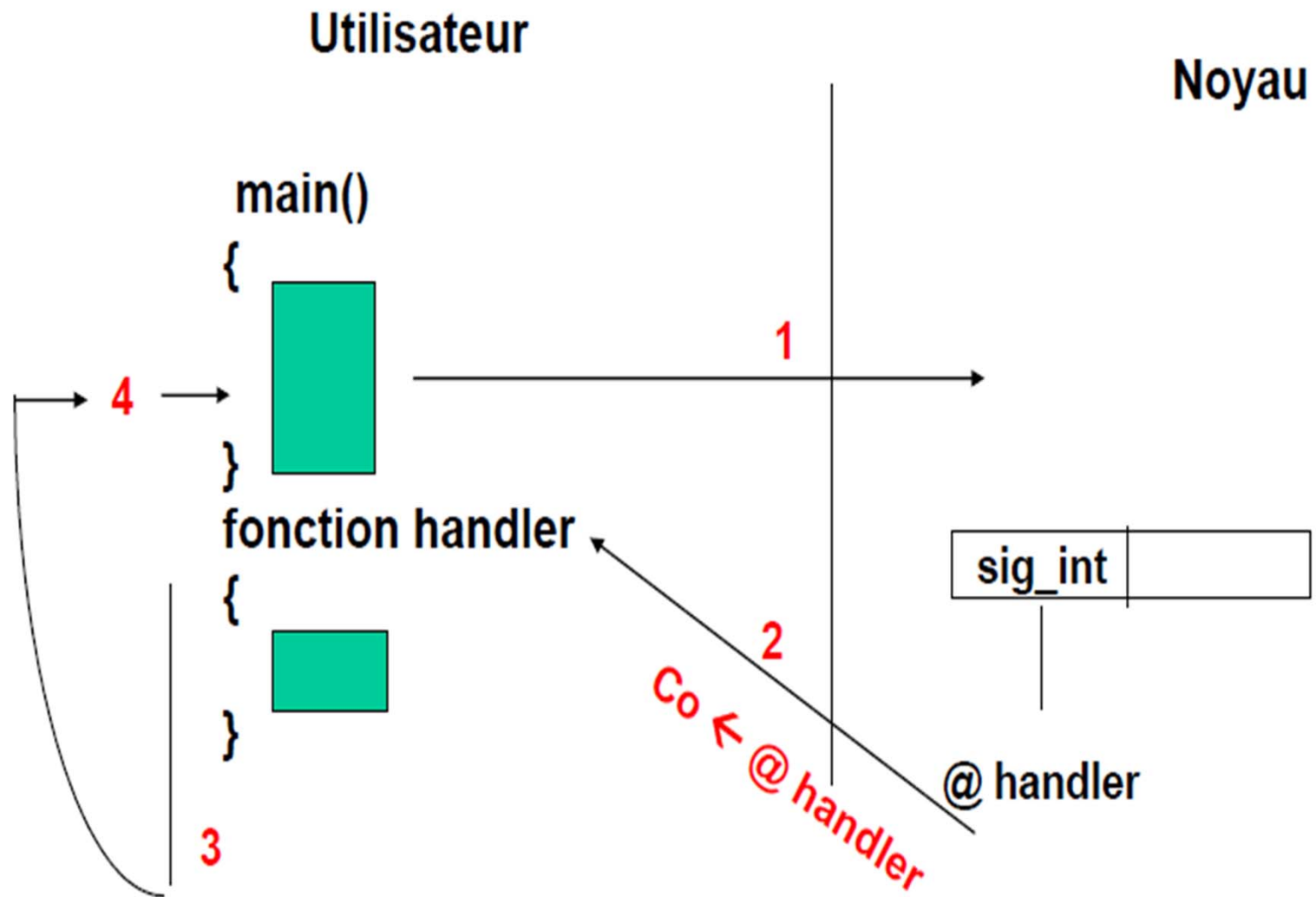
- Quand un processus passe du mode noyau au mode utilisateur (quand il a terminé un appel système). Ce veut dire que le traitement du signal est différé tant que le processus n'est pas revenu en mode utilisateur.
- Avant de passer en état bloqué
- En sortant de l'état bloqué

Si c'est le cas : le processus réagit en:

- ✓ Exécutant le gestionnaire associé
- ✓ Ignorant le signal ou
- ✓ Se terminant

Après l'exécution du gestionnaire, le processus reprendra le code interrompu à l'instruction qui suit juste avant le gestionnaire.

Les signaux : Réception d'un signal



Les signaux : Capture d'un signal

▲ Pour installer un handler, on utilise

```
#include <signal.h>
typedef void (*sig_handler_t)(int);
sig_handler_t signal ( int signum,  sig_handler_t handler );
```

- Le premier paramètre est le numéro ou le nom du signal à capturer
- Le second est la fonction gestionnaire à exécuter à l'arrivée du signal.
- SIG_DFL désigne l'action par défaut et
- SIG_IGN indique au système que le processus appelant veut ignorer le signal.
- signal retourne le gestionnaire précédent ou SIG_ERR en cas d'erreur.

Les signaux : Capture d'un signal

△ Pour installer un handler, on utilise

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

```
int sigaction( int sig , const struct sigaction *act , struct sigaction * old_act);
```

△ La structure sigaction a la forme suivante:

```
struct sigaction
```

```
{  
    void (* sa_handler)( int);      /*le gestionnaire*/  
    int sa_flags; /*options*/  
    sigset_t sa_mask; /* Le masque des signaux à bloquer durant l'exécution  
    du gestionnaire*/  
}
```

△ SIG_DFL : traitant par défaut.

△ SIG_IGN : ignorer le signal.

Les signaux : Attente d'un signal

L'appel système `pause()` suspend l'appelant jusqu'au prochain signal.

```
#include<unistd.h>
int pause(void);
```

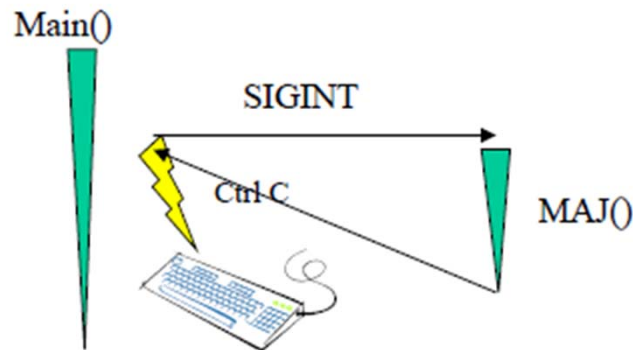
L'appel système `sleep(v)` suspend l'appelant jusqu'au prochain signal ou l'expiration de `v` secondes.

```
#include<unistd.h>
int sleep(v);
```

Les signaux : Exemple(1)

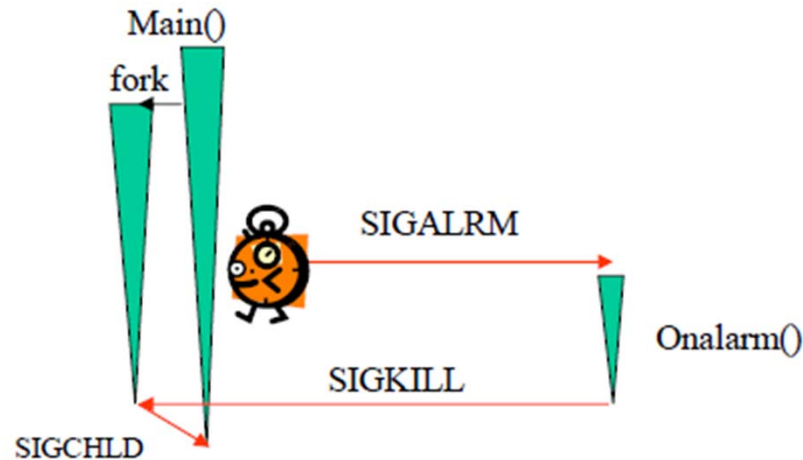
```
#include<stdio.h>
#include<signal.h>
#include<sys/types.h>
#include<unistd.h>

void MAJ()
{
    printf("\n Handler reçu \n");
    printf("AZERTY \n");
}
```



```
main()
{
    int i;
    signal (SIGINT,MAJ);
    i=0;
    while (i<50)
    {
        printf("\n azerty");
        sleep(5);
        i++;
    }
    printf("\n fin du processus
normal\n");
}
```

Exemple 2



```
#include <sys/types.h>
#include <unistd.h>
#include <stdio.h>
#include <signal.h>
pid_t pid;
```

```
void onalarm ()
{
    printf("Handler
    onalarm\n");
    printf("je suis le fils de PID
    %d, je mort!!!! \n",pid);
    kill (pid, SIGKILL);
}
```

Exemple 2: suite

```
main()
{
    pid = fork();
    if (pid == -1)
        printf ("erreur creation de processus");
    else
        if (pid == 0)
        {
            printf ("\n valeur du fork %d", pid);
            printf (" \n je suis le fils, mon pid est %d \n",
                getpid());
            printf("je vais dormir pendant 50 secondes !!!! \n");
            sleep(50);
        }
}
```

Exemple 2: suite

```
else
{
printf ("je suis le père \n");
printf ("je vais tuer mon fils de PID %d apres 2
secondes \n",pid);
signal (SIGALRM, onalarm);
alarm(2);
wait (); }
}
```

Exemple 3

```
// test_signaux.c
#include <signal.h>
#include <stdio.h>
#include <unistd.h>
static void action(int sig)
{
    printf("On peut maintenant m'eliminer\n");
    signal(SIGTERM, SIG_DFL);
}

int main()
{
    if( signal(SIGTERM, SIG_IGN) == SIG_ERR)
        perror("Erreur de traitement du code de l'action\n");
    if( signal(SIGUSR2, action) == SIG_ERR)
        perror("Erreur de traitement du code de l'action\n");
    while (1)
        pause();
}
```