
# Fondamentaux des Applications Réparties (FAR)

**Chouki Tibermacine**
**Chouki.Tibermacine@umontpellier.fr**
**Diapos de D. Delahaye**


POLYTECH®
MONTPELLIER

# Plan de l'ECUE

- Partie système :
  1. Threads et problèmes de communication inter-processus ou -threads
  2. Communication par tubes et signaux
- Partie réseaux :
  1. Modèle OSI et couches réseaux basses
  2. Protocole de la couche réseau (IP)
  3. Protocoles de de la couche transport (TCP & UDP)
  4. Programmation réseau en langage C (sockets)
  5. Programmation par RPC (*Remote Procedure Call*)
  6. Programmation par Websockets (sockets sur HTTP)
  7. Programmation par objets distribués (Java RMI)

# Introduction

## Objectifs de ce cours

- Savoir gérer les tubes (« pipes ») et signaux Unix ;
- Savoir les utiliser pour la communication entre plusieurs processus.

## Langage support

- Utilisation du langage C ;
- Langage dans lequel est codé le noyau des systèmes Unix ;
- Langage de prédilection pour faire de la programmation système.

# Les tubes

## Qu'est-ce que c'est ?

- Canaux de communication unidirectionnels ;
- Manipulés à l'aide de descripteurs de fichiers ;
- Accessibles en lecture ou écriture.

## Qui accède à un tube ?

- Le créateur du tube lui-même ;
- Les processus descendants du créateur du tube.

# Les tubes -suite-

## Comment se passe la lecture ?

- Un tube ≡ FIFO (« First In First Out ») ;
- Toute lecture est destructrice.

## Primitives pour les tubes

```
#include <unistd.h>
int pipe (int descripteur [2]);
int write (int descripteur, char *buffer, int longueur);
int read (int descripteur, char *buffer, int longueur);
int close (int descripteur);
```

# Primitives pour les tubes

## Création

pipe(descripteur) :

- Crée un tube et retourne 0 en cas de succès, -1 en cas d'échec ;
- descripteur[0] : descripteur du tube créé en lecture ;
- descripteur[1] : descripteur du tube créé en écriture.

# Primitives pour les tubes

## Écriture

write(descripteur, buffer, longueur) :

- Écrit dans le tube ayant `descripteur` pour descripteur en écriture ;
- Écrit `longueur` octets, commençant en mémoire à l'adresse `buffer` ;
- Renvoie le nombre d'octets écrits ou -1 en cas d'échec ;
- Exemple : write(p[1], &v, sizeof(float)), où v variable de type float.

# Primitives pour les tubes

## Lecture

read(descripteur, buffer, longueur) :

- Lit dans le tube ayant descripteur pour descripteur en lecture ;
- Lit longueur octets (au maximum) et les place à l'adresse buffer ;
- Renvoie le nombre d'octets lus, ou 0 si le tube est vide et qu'il n'y a plus d'écrivains sur le tube (s'il reste des écrivains, le processus se bloque en attente d'information), ou enfin -1 en cas d'erreur ;
- Exemple : read(p[0], &v, sizeof(float)), où v variable de type float.

# Primitives pour les tubes

## Fermeture

close(descripteur) :

- Ferme le tube en lecture ou en écriture (en fonction du descripteur passé en argument), pour le processus ayant appelé close seulement
- Renvoie 0 en cas de succès ou -1 en cas d'échec.

# Tubes et fork

## Principe du tube

- Un processus écrit dans descripteur[1] ;
- Un autre processus lit dans descripteur[0] ;
- Mais la création se fait dans un processus ;
- Comment l'autre processus devine-t-il les valeurs de descripteur ?

## Solution : fork des processus

- Duplication des processus ;
- Processus « lourds » (copie de la mémoire) ;
- Primitive : fork.

# Tubes et fork

## Primitive fork

```
pid_t fork(void);
```

- Retourne le pid du fils dans le processus père ;
- Retourne 0 dans le processus fils ;
- Le processus fils est une copie conforme de son père.

## Quelques primitives utiles

- `getpid()` : retourne le pid du processus courant ;
- `getppid()` : retourne le pid du processus père ;
- `wait()` : suspend le processus jusqu'à ce qu'au moins un de ses processus fils se termine (synchronisation).

# Exemple : Ouverture et écriture sur un tube

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

int main(void) {
    pid_t pid;
    int descr[2];
    char buffer[256];
    pipe(descr);
    pid = fork();
    if (pid != 0) { /* Processus pere */
        sprintf(buffer, "Message du pere");
        write(descr[1], buffer, 256);
    }
    return EXIT_SUCCESS;
}
```

# Exemple : Lecture sur un tube

```c
int main(void) {
    pid_t pid;
    int descr[2];
    char buffer[256];
    pipe(descr);
    pid = fork();
    if (pid != 0) { ... }
    else if (pid == 0) { /* Processus fils */
        read(descr[0], buffer, 256);
        printf("Message recu = %s\n", buffer);
    }
    return EXIT_SUCCESS;
}
```

# Exemple : Fermeture d'un tube

```c
int main(void) {
    pid_t pid;
    int descr[2];
    char buffer[256];
    pipe(descr);
    pid = fork();
    if (pid != 0) { /* Processus pere */
        close(descr[0]);
        sprintf(buffer, "Message du pere");
        write(descr[1], buffer, 256);
    }
    ...
```

# Exemple : Fermeture d'un tube

```c
int main(void) {
    ...
    else if (pid == 0) { /* Processus fils */
        close(descr[1]);
        read(descr[0], buffer, 256);
        printf("Message recu = %s\n", buffer);
    }
    return EXIT_SUCCESS;
}
```

## Important

- Fermer l'entrée du tube qui lit ;
- Fermer la sortie du tube qui écrit.

# Exercices

Écrire des programmes C pour les exercices suivants :

- On a un processus père et un processus fils :
  - Le processus fils demande la saisie d'un entier, l'envoie au processus père, puis demande si l'utilisateur veut effectuer une autre saisie, etc. ;
  - Le processus père fait la moyenne de tous les entiers reçus et l'affiche lorsqu'il reçoit le dernier entier.
- On a un processus père et un processus fils :
  - Le processus père envoie 5 entiers au processus fils ;
  - Le processus fils incrémente ces entiers et les renvoie au processus père ;
  - Le processus père reçoit ces 5 entiers incrémentés et les affiche.

# Exercices -suite-

- On veut écrire la fonction puissance $x^n$ récursivement et par dichotomie :
  - $x^n = x^{n/2} \times x^{n/2}$ si $n$ est pair
  - ou $x^n = x \times x^{(n-1)/2} \times x^{(n-1)/2}$ si $n$ est impair
  - L'expression $x^{n/2}$ (ou $x^{(n-1)/2}$) sera calculée récursivement par un processus fils, puis le résultat sera envoyé au processus père qui calculera $x^n$. Le cas de base est $n = 0$.

# Les signaux

## Qu'est-ce que c'est ?

- Mécanisme asynchrone permettant d'informer un processus que quelque chose s'est produit dans le système.

## D'où viennent-ils ?

- Émis par le noyau (division par zéro, overflow, etc.) ;
- Émis depuis le clavier par l'utilisateur (<Ctrl-Z>, <Ctrl-C>, etc.) ;
- Émis depuis le shell par la commande kill ;
- Émis par la primitive kill dans un programme C.

# Les signaux -suite-

## Comment ça marche ?

- Quand un signal est envoyé à un processus, le système d'exploitation interrompt l'exécution normale de celui-ci ;
- Si le processus possède une routine de traitement pour le signal reçu, il lance son exécution, sinon il exécute une routine par défaut.

# Émission d'un signal

## Primitive kill

```
#include <signal.h>
int kill (pid_t pid, int num);
```

- pid est l'identifiant du processus destinataire du signal ;
- num est le numéro du signal à envoyer ;
- Renvoie 0 en cas de succès ou -1 en cas d'échec ;
- Liste des signaux supportés par Linux : kill -l, man 7 signal.

# Émission d'un signal -suite-

## Primitive raise

```
int raise (int num);
```

- raise(signal) ≡ kill(getpid(), signal) ;
- le processus s'envoie à lui-même un signal ;
- Pseudo-mécanisme de gestion des exceptions.

# Exemple : Tuer son fils

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <signal.h>

int main(void) {
    pid_t pid = fork();
    if (pid != 0) { /* Processus pere */
        char rep;
        do {
            printf("Voulez—vous tuer mon fils (o/n) ? ");
            scanf("%c", &rep);
        }
        while (rep != 'o');
        kill(pid, SIGTERM);
        printf("Fils tue !\n");
    }
    ...
```

# Exemple

## Tuer son fils

```
    ...
    else if (pid == 0) { /* Processus fils */
      while (1) {
        printf("Je suis le fils un peu penible\n");
        sleep(1);
      }
    }
    return EXIT_SUCCESS;
}
```

# Réception et traitement d'un signal

## Primitive signal

```
signal(num, traitement);
```

- traitement = SIG_DFL : traitement par défaut ;
- traitement = SIG_IGN : ignorer le signal ;
- traitement = fonction (qui prend le numéro du signal en entrée) : traitement spécifique utilisateur.

# Exemple : Résister au <Ctrl-C>

```c
#include <stdio.h>
#include <stdlib.h>
#include <signal.h>

void traitement (int n) {
    printf("Ton signal %d est inutile !\n", n);
}

int main(void) {
    signal(SIGINT, traitement);
    printf("Essaie de me tuer...\n");
    for(;;) {
      printf("Essaie encore...\n");
      sleep(1);
    }
}
```

# Les zombies, c'est mal !

## Exemple d'un programme qui crée des zombies

```c
// Afficher les zombies avec : top -b | grep Z
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
int main(void) {
  pid_t pid; int i;
  for (i = 0; i < 5; i++) {
    pid = fork();
    if (pid != 0) /* Processus pere */
      printf("Processus pere (%d)\n", i);
    else if (pid == 0) { /* Processus fils */
      printf("Processus fils (%d)\n", i);
      exit(0);
    }
  }
  for (;;);
  return EXIT_SUCCESS;
}
```

# Les zombies, c'est mal !

## Exemple d'un programme qui évite de créer des zombies

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/wait.h>
int main(void) {
  pid_t pid;  int i, status;
  for (i = 0; i < 5; i++) {
    pid = fork();
    if (pid != 0) { /* Processus pere */
      printf("Processus pere (%d)\n", i);
      wait(&status); // Attend le processus fils
    }
    else ...
  }
  for (;;);
  return EXIT_SUCCESS;
}
```

# Exercice

## Écrire les programmes C de l'exercice suivant

- On a un processus père et un processus fils :
  - Le processus père demande en boucle la saisie d'un entier, et l'envoie au processus fils ;
  - Le processus fils teste l'entier qu'il reçoit : s'il est négatif, il envoie le signal utilisateur SIGUSR1 au processus père ;
  - Le processus père met en place une procédure de traitement s'il reçoit le signal SIGUSR1, qui affiche que l'entier est négatif.