# L3 Informatique : Cours Systèmes et Réseaux
# Communications inter-processus

Olivier Togni
Université de Bourgogne, IEM/LIB
Bureau G206
olivier.togni@u-bourgogne.fr

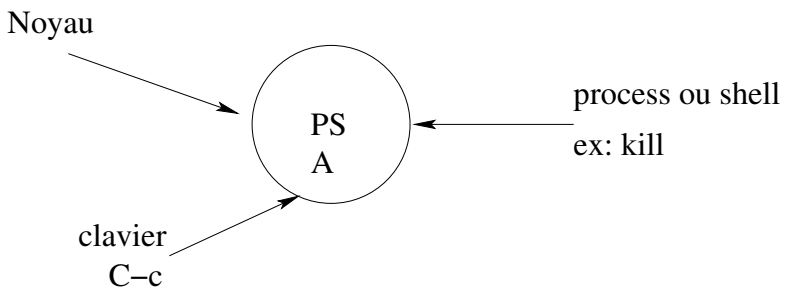
11 octobre 2023

# Plan

1. Signaux Unix
   - principes
   - signaux en shell et C
   - signaux temps réél
2. Tubes
   - tubes anonymes
   - tubes nommés
3. Sockets
   - principes
   - rappel du fonctionnement des réseaux IP
   - interface des sockets en C

# Signaux

Signal = outil de base de la notification d'évènement.
Fonctionnent sur un schéma essentiellement asynchrone.




De façon simplifié, quand le signal arrive, le noyau interrompt le processus qui déclanche alors son gestionnaire de signaux (handler) pour réagir de façon appropriée.
Cependant une primitive (sigpause) permet à un processus de se mettre en attente d'un signal.

# Signaux

Les signaux utilisés initialement plutôt pour la gestion d'erreurs ont vu leur rôle s'étendre dans les versions plus récentes d'Unix. Ils ont néanmoins de sérieuses limitations :

- Ils coûtent cher : ils sont lancés par un appel système, le récepteur est interrompu, sa pile à l'exécution est modifiée, le handler prend la main, la pile à l'exécution est restaurée.
- En nombre limité : de l'ordre de 20 à 60, dont seulement deux signaux pour la programmation utilisateur (les autres signaux sont utilisés par le système lui même).
- Ils sont strictement limités au rôle de notification d'évènement: ne peuvent pas apporter d'information complémentaire (au moins pour les signaux utilisateurs), ni même l'identité de l'émetteur.

# Les signaux sous Unix

Les signaux prédéfinis sont identifiés par un nom et un numéro. Le nom est commun à toutes les versions d'UNIX, le numéro peut varier d'un système à l'autre.
Chaque signal est associé à un comportement par défaut (ce que doit faire le processus cible à réception de ce signal):
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)

# Principaux signaux

| num | nom | comportement par défaut | modifiable ? |
|---|---|---|---|
| 1 | SIGHUP | envoyé lors de la déconnexion | non |
| 2 | SIGINT | utilisation de la touche INTR (DEL) | oui |
| 3 | SIGQUIT | utilisation de la touche QUIT (CTR \) | oui |
| 4 | SIGILL | opération illégale | non |
| 5 | SIGTRAP | après chaque instruction en mode trace | non |
| 8 | SIGFPE | erreur opération en virgule flottante | non |
| 9 | SIGKILL | destruction inconditionnelle du processus | non |
| 11 | SIGSEGV | violation de segment mémoire | non |
| 12 | SIGSYS | erreur de paramètre dans un appel système | non |
| 13 | SIGPIPE | erreur dans un pipe (écriture sans lecteur) | non |
| 14 | SIGALARM | signal d horloge | non |
| 15 | SIGTERM | indicateur de fin normale d'un processus | non |
| 16 | SIGUSR1 | rien : à disposition de l'utilisateur | oui |
| 17 | SIGUSR2 | rien : à disposition de l'utilisateur | oui |

Pour les programmes utilisateurs, on dispose des signaux SIGUSR1 et SIGUSR2.

# Les signaux en shell

Pour la manipulation des signaux, on dispose des opérations suivantes :
Envoi de signaux par

```
kill -s signal pidDestinataire
```

Pour modifier le comportement par défaut d'un signal on utilise la primitive :

```
trap 'instructions' signaux
```

Les instructions de la liste sont séparées par des points virgules. Si la liste des instructions est vide, le signal est ignoré. S'il n'y a pas de liste, on revient au comportement par défaut.

# Exemple

```
P1
#!/bin/sh
trap 'echo fini;exit 1' USR1
P2 $PID&
while :
do
echo "ca boucle"
done

P2
#!/bin/sh
kill -s USR1 $1
```

# Remarque

La commande `trap -l` donne le liste des signaux auxquels peux répondre trap.
Pour bloquer un processus père jusqu'à la fin de l'un de ses fils :

```
wait pidDuFils
```

Sans paramètre, le wait fait attendre la fin de tous les fils.

# Les signaux en C

- signal : ancien, simple, peu portable, non fiable
- sigaction : plus complexe mais standard
- sigqueue: pour signaux temps-réél

# L'interface des signaux non fiables (signal)

Le traitement en C des signaux non fiables repose sur deux fonctions `signal` et `kill` (prototype dans `signal.h`).
Mise en place (et réinitialisation) du gestionnaire:

```
int signal(signal, *fonction a utiliser)
```

Pointeurs prédéfinis sur des fonctions de traitement de signaux : `SIG_IGN` pour ignorer le signal et `SIG_DFL` pour revenir au comportement par défaut.
Envoi du signal:

```
int kill(pidDest, signal)
```

Envoi possible si le process appelant en a les droits (UID émétteur = UID récepteur)
Si pid==0, envoi à tous les processus de même GPID que le process appelant

# Exemple : masquage du ctrl-c

```
#include <stdio.h>

void sigproc(){
 signal(SIGINT, sigproc);
     printf("Vous avez tapé ctrl-c \n") ;
}

void quitproc()
{
     printf("Vous avez tapé ctrl-\, on quitte\n") ;
     exit(0) ;
}
```

# Exemple : masquage du ctrl-c

```
main(){
    signal(SIGINT, sigproc) ;
    signal(SIGQUIT, quitproc) ;
    printf("ctrl-c masqué\n") ;
    for( ; ;) ; /* boucle infinie*/
}
```

# Attente et temporisation

Mise en attente du processus tant qu'un signal n'est pas reçu :

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

Armer une temporisation :

- Indiquer une durée
- Après cette durée, un signal SIGALARM sera reçu
- L'exécution se poursuit entre temps
- Application : avant appel bloquant (lecture clavier)

```
#include <unistd.h>
int alarm(int nbsec);
```

# L'interface des signaux fiables (sigaction)

Les signaux fiables reposent sur des jeux de signaux (définis en C par le type `sigset_t`). Un jeu de signaux est manipulé par des fonctions (prototypes dans `signal.h`) qui permettent :

- d'initialiser le jeu de signaux (`sigemptyset`),
- de remplir le jeu avec autant de signaux que possible (`sigefullset`),
- d'ajouter un signal à un jeu (`sigaddset`),
- de supprimer un signal d un jeu (`sigdelset`),
- vérifier si un signal fait partie d un jeu (`sigismember`).

Action associée à un signal:

```
int sigaction(int signum, const struct sigaction *act,
              struct sigaction *oldact)
```

# Signaux temps-réel

- signaux classique : si un signal arrive pendant le traitement d'un signal ?
- si c'est le même signal ?
- signaux temps réel : file d'attente (1024), ordre préservé, information supplémentaire
- numéros 32 à 63, Posix 1b, noms de SIGRTMIN à SIGRTMAX, ex : SIGRTMIN+2

# Les tubes

Les tubes permettent de faire transiter des informations (messages) entre deux processus
Sous Unix les tubes sont orientés
Assimilés à des fichiers séquentiels de caractères :

- données transmises dans l'ordre (FIFO)
- sans perte (sauf sockets...)
- lecture $n$ octets: bloquante si pas assez d'octets à délivrer, sauf si fin de fichier
- écriture de $p$ octets: bloquante si manque de place (taille limite)
- fin de fichier quand plus aucun processus n'accède au tube en écriture

Plusieurs types de tubes en shell et C: tubes anonymes (popen, pipe), tubes nommés (mkfifo), tubes étendus entre machines distantes sur le réseau (sockets).

# Tubes anonymes popen

ouverture d'une communication avec un autre processus créé pour exécuter une commande. Le processus appelant peut se positionner soit en lecture sur le tube (il récupère alors les résultats de la commande), soit en écriture (il envoie des données à la commande).
Le descripteur renvoyé est de type `FILE *`. Lecture et écriture avec `fread` et `fwrite`.

```
FILE *popen(char *command, char *type) ;
int pclose(FILE *stream) ;
```

# Tubes anonymes pipe

La fonction `pipe` permet de créer deux descripteurs d'accés à un tube. Les descripteurs sont dans un tableau de deux entiers (le premier permet de lire, le second d'écrire).
Le prototype est dans `unistd.h`. Manipulations par read, write et close.
On utilise souvent pipe avant un fork pour permettre aux deux processus de communiquer.

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

# Pipe : utilisation

```
int main(void) {
        int     fd[2], nb;
        pid_t   childpid;

        pipe(fd);
        if((childpid = fork()) == -1) erreur("fork");
        if(childpid == 0)
        {  // dans le fils : écriture dans le tube
                close(fd[0]);
                write(fd[1], string, (strlen(string)+1));
                close(fd[1]);
                exit(0);
        }
        else { // dans le père : lecture dans le tube
                close(fd[1]);
                nb=read(fd[0],buff,sizeof(buff));
                wait(NULL);  // attente de fin du fils
        }
        return(0);
```

# Principes des tubes nommés

Un tube nommé est un fichier spécial (de type 'p = pipe) qui a un nom et un chemin dans le SF)
Ouvert par les deux processus ayant besoin de communiquer (l'un ouvrant en lecture et l'autre en écriture).
En shell : `mkfifo nomFic`

```
#!/bin/bash

mkfifo /tmp/canal
echo "Bonjour" >/tmp/canal
read line </tmp/canal
echo "Reçoit $line"
```

```
#!/bin/bash

read line </tmp/canal
echo "Reçoit $line"
echo "Au revoir" >/tmp/canal
```

# Lecture/écriture en boucle

On fait souvent des envois/réceptions dans des boucles
Attention, certaines situations peuvent être problématiques

## Exemple

```
P1
#!/bin/bash
mkfifo montube
while true; do
echo "oui" >montube
done
```

```
P2
#!/bin/bash

while true; do
read line <montube
done
```

S'exécute normalement, sauf si les primitives open, read/write, close correspondantes aux lignes echo et read sont entrelacées,
par ex : P1.open, P2.open, P1.write, P1.close, P1.open, P2.read, P2.close, P1.write ⇒ plantage (silencieux) de P1 avec SIGPIPE

# Tubes nommés mkfifo en C

La fonction `mkfifo` permet de créer un support de communication avec un autre processus quelconque en créant Manipulation par open, read, write, close.

```
mkfifo(char * chemin, int droit) ;
```

Exemple : l'instruction `mkfifo(‘‘SR/PG/TESTmkfifo’’,0666);` produit le fichier :

```
9 prw-r r 1 mnt root 0 Mar 26 10 :50 TESTmkfifo 14 15 4.49
```

# Les sockets

Socket = API permettant de faire communiquer des processus en réseau

- Apparues en 82 sur BSD4.1c
- Socket = prise = extrémité d'un canal de communication entre 2 ou plusieurs processus. Ce point est représenté par un variable entière (descr) similaire à un descripteur de fichier.
- caractérisée par un type et un domaine
- Généralisation du mécanisme d'E/S d'Unix : même type de mécanisme, mais spécificités dues aux caractéristiques un peu spéciales des E/S réseau:
  - non symetrie de relation cli/serv
  - connexion de type connecté ou non, définie par quintuplet: (proto, adr locale, process local, adr éloignée, process éloigné)
  - l'interface doit permettre d'accéder à différents protocoles (TCP, UDP, XNS, ...pas spécifique à TCP/IP)

# Internet c'est quoi ?

Basé sur une architecture TCP/IP du nom des deux principaux protocoles utilisés

Provient du réseau Arpanet de la Défense Américaine (années 70)

Internet = des millions de réseaux interconnectés par des noeuds d'interconnexion (routeurs).

| |
|---|
| 7 Application |
| 6 Présentation |
| 5 Session |
| 4 Transport |
| 3 Réseau |
| 2 Liaison |
| 1 Physique |

Modèle OSI

| |
|---|
| Application: *SSH, HTTP, Telnet, DNS, ...* |
| Transport: *TCP, UDP* |
| Inter−réseau: *IP* |
| Accès au réseau: *Ethernet, TokenRing, FDDI, PPP,...* |

Architecture TCP/IP

# Adresses

Chaque interface (d'accès au réseau) possède une adresse physique inscrite sur la carte (adr MAC 48bits pour carte Ethernet, unique au monde).

Au niveau IP(v4): adresses de 32 bits $= 4 \times 1$ octet représenté en décimal pointé.
Ex: 192.52.237.36

Version 6 : adresses IPv6 sur 128bits, par exemple :
2a03:2880:f030:f:face:b00c:0:2

cmd `ifconfig` pour voir les adresses des interfaces

# Adresses IPv4 et sous réseaux

chaque adresse = prefixe_réseau + id_machine 4 classes d'adresses: A pref=8, B pref=16, C pref=24, D multicast + E reservé

## Exemple

192.52.237.36 est une adr de classe C, pref_res=192.52.237, id_machine=36
192.52.237.0 = adr du réseau
192.52.237.1 ... 254 = adr possibles pour les machines
192.52.237.255 adr de diffusion (broadcast) sur le réseau

# Format d'un paquet IPv4

```
 0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|Version|  IHL  |Type of Service|          Total Length         |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|         Identification        |Flags|      Fragment Offset    |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|  Time to Live |    Protocol   |         Header Checksum       |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                       Source Address                          |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                    Destination Address                        |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                    Options                    |    Padding    |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

# IP et le routage

- Le protocole IP (Internet Protocol) s'occupe du routage des paquets (chemin qui va être emprunté de la source à la destination).
- Pas de contrôle d'erreur ni perte de paquet => service minimum.
- Chaque paquet est routé indépendament des autres (commutation de paquets). L'en-tête du paquet (20 octets) contient les adr IP src et dst.
- Chaque routeur (et hôte) possède une table de routage et en fonction de l'adr dst, il choisit l'entrée qui correspond le mieux et réémet le paquet sur l'interface en sortie correspondante.

# Commandes

`route` ou `ip` pour voir la table de routage

```
otogni@otogni-XPS-13-7390:~/Bureau$ route -nv
Table de routage IP du noyau
Destination     Passerelle      Genmask         ...  Iface
0.0.0.0         192.168.0.254   0.0.0.0         ...  wlp2s0
169.254.0.0     0.0.0.0         255.255.0.0     ...  wlp2s0
192.168.0.0     0.0.0.0         255.255.255.0   ...  wlp2s0
```

`ping adrIP ou nom` pour tester l'accessibilité d'une station
`traceroute adrIP ou nom` pour voir le chemin jusqu'à la destination

# Protocoles de transport: TCP et UDP

Au niveau transport, l'en-tête des paquets contient des champs numéro de port src et dst qui servent à indiquer à quelle application est destinée le paquet.
Les numéros $< 1024$ sont affectés à des applicationss standard. Ex: HTTP=80, SSH=22, ...

**UDP** (User Datagramm Protocol): service de transport non fiable en mode non connecté = service mini IP + num de port

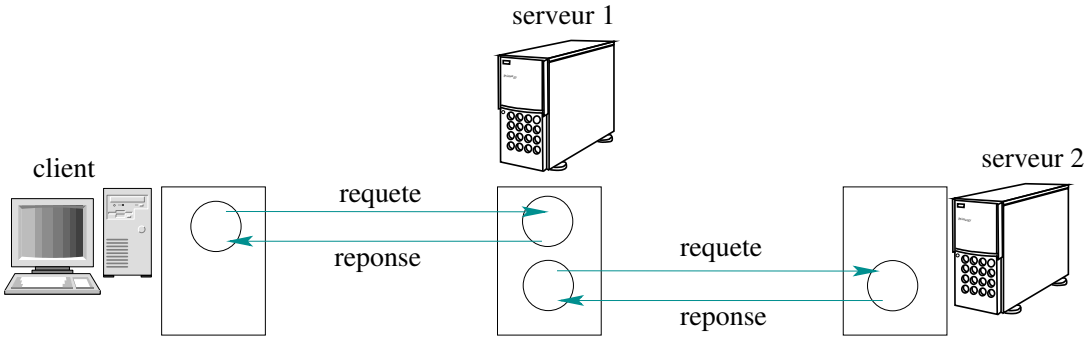
**TCP** (Transmission Control Protocol): service de transport fiable en mode connecté (sur IP non fiable, sans connexion) =>

- établissement, maintien et fermeture d'une connexion virtuelle
- acquitements, séquencement et réassemblage des données
- contrôle de flux par fenêtre glissante

# Protocoles applicatifs

Fonctionnent presque tous sur le mode client/serveur: client et serveur sont deux processus situés le plus souvent sur des machines différentes.

- Le serveur tourne en parmanence (en arrière plan: on parle de démon en Unix).
- Le client envoie une ou plusieurs requêtes au serveur qui répond par une ou plusieurs requêtes.
- Un serveur peut être client d'un autre serveur.



# Protocoles applicatifs

Il en existe des milliers, certains publics (RFC IETF), d'autres privés (ex : skype)

- Protocoles de gestion : NIS, LDAP (admin centralisée des logins et groupes), NFS (montage de SF éloignés), DNS (traduction nom d'hôte - addr IP), SMTP (service d'envoi des mails), POP et IMAP (consultation de boites aux lettres distantes), SNMP (admin réseau à distance), etc
- Protocoles d'accès distant : TELNET, SSH, SFTP, scp
- protocoles multimédia: SIP (gestion de session), RTP/RTCP (transport de flux tps réel), etc

# Création et fermeture d'une socket

**Création d'une socket:**

```
int sd=socket(int domain, int type, int proto)
```

domaine = AF_UNIX (domaine local), AF_INET (IPv4), AF_INET6 (IPv6), ...

type = SOCKSTREAM, SOCKDGRAM, SOCKRAW, ...
proto= IPPROTO_UDP, IPPROTO_TCP, ... mais souvent mis à 0 car couple domaine/type fixe le proto:
AF_INET + SOCKSTREAM = TCP
AF_INET + SOCKDGRAM = UDP
AF_INET + SOCKRAW = IP

**Fermeture:** `close(sd)`

# Association d'un port et d'une adresse IP à la socket

`int bind(int sd, struct sockaddr *monAddr,socklen_t tailleAddr)`

structure générique sockaddr, puis deux variantes:

```
struct sockaddr_un {
short sun_family; // AF_UNIX
char sun_path[128]; // chemin d'accès du fichier
}

struct sockaddr_in {
short sin_family; // AF_INET
unsigned short sin_port; // num port
struct in_addr sin-addr; // IP 32 bits
char sin_zero[8]; // inutilisé
}

struct in-addr {
unsigned long s_addr; // 32 bits
}
```

# Connexion et envoi des données

`int connect(int sd, struct sockaddr *serveurAddr, socklen_t tailleAddr)`

- fait par client pour mode connecté
- **Envoi de données:** primitives write, send, sendto, sendmsg
  sendto et sendmsg réclament l'adr dst à chaque envoi
- **Réception:** read, readv, recv, recvfrom, recvmsg
- **File d'attente:** `int listen(int sf, int tailleFile)`
  Met en file d'attente les demandes de connexion provenant des clients

# Accepter un connexion

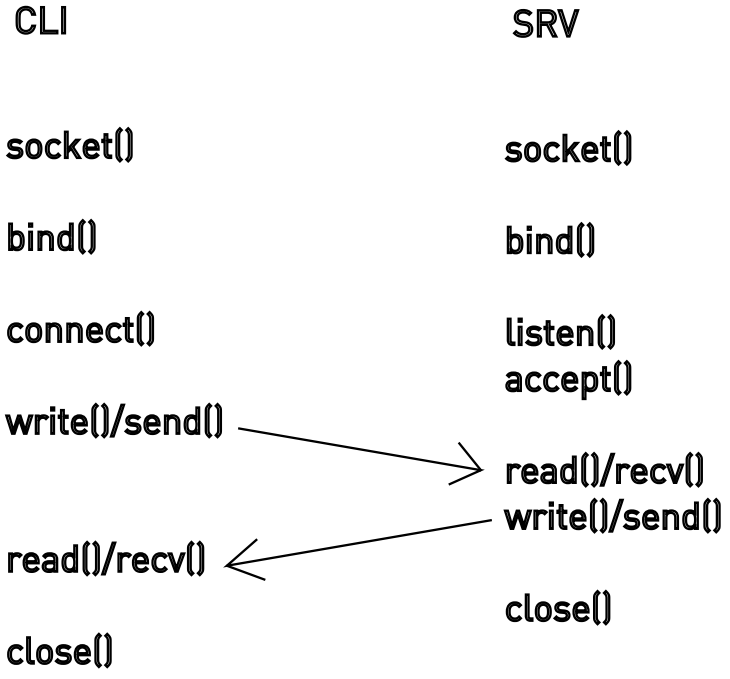
- Quand le serveur invoque `accept`, il se met en attente bloquante d'une connexion.

```
int accept (int sd, struct sockaddr *serveurAddr,
socklen_t tailleAddr)
```

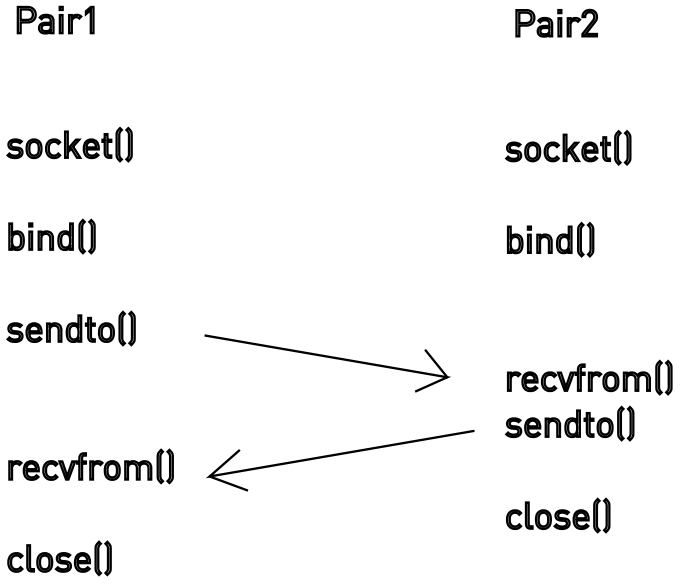
- retourne le descr d'une nouvelle sock utilisée pour dialoguer avec client et remplit la struct avec addr du client.

# Schéma en mode connecté/non connecté

## Mode connecté




## Mode non connecté



# Limitations

- Primitives de lectures/ecriture sont bloquantes par défaut (on peut les rendre non bloquantes à l'aide de la primitive fcntl()).
- Donc si le serveur attend des données sur un descripteur, il ne peut rien faire d'autre pendant ce temps.

# Serveur récursif

- Avec duplication de processus (fork()) ou processus légers (threads)
- primitives de scrutation de descripteurs de fichier: select (BSD) et poll(S V)
- select permet de surveiller un ensemble de descripteurs :
  - Si aucun actif, process endormi, pas de ressources CPU utilisées, dès qu'un descr devient actif, le noyau reveille le process et select se termine avec infos sur le descr qui a provoqué le reveil.
  - On doit: 1) remplir une struct fd_set avec les descr a surveiller, 2) appel au select 3) tester chacun des descripteurs
- poll() propose les mêmes fonctionnalités, mais avec une approche légèrement différente: on remplit un tab de struct pollfd, puis appel a poll sur le tab, puis on regarde si données disponibles sur chaque descripteur