# Programmation système & Systèmes d'exploitation

Deuxième partie

HEPIA
Mickaël Hoerdt

# Objectifs du cours

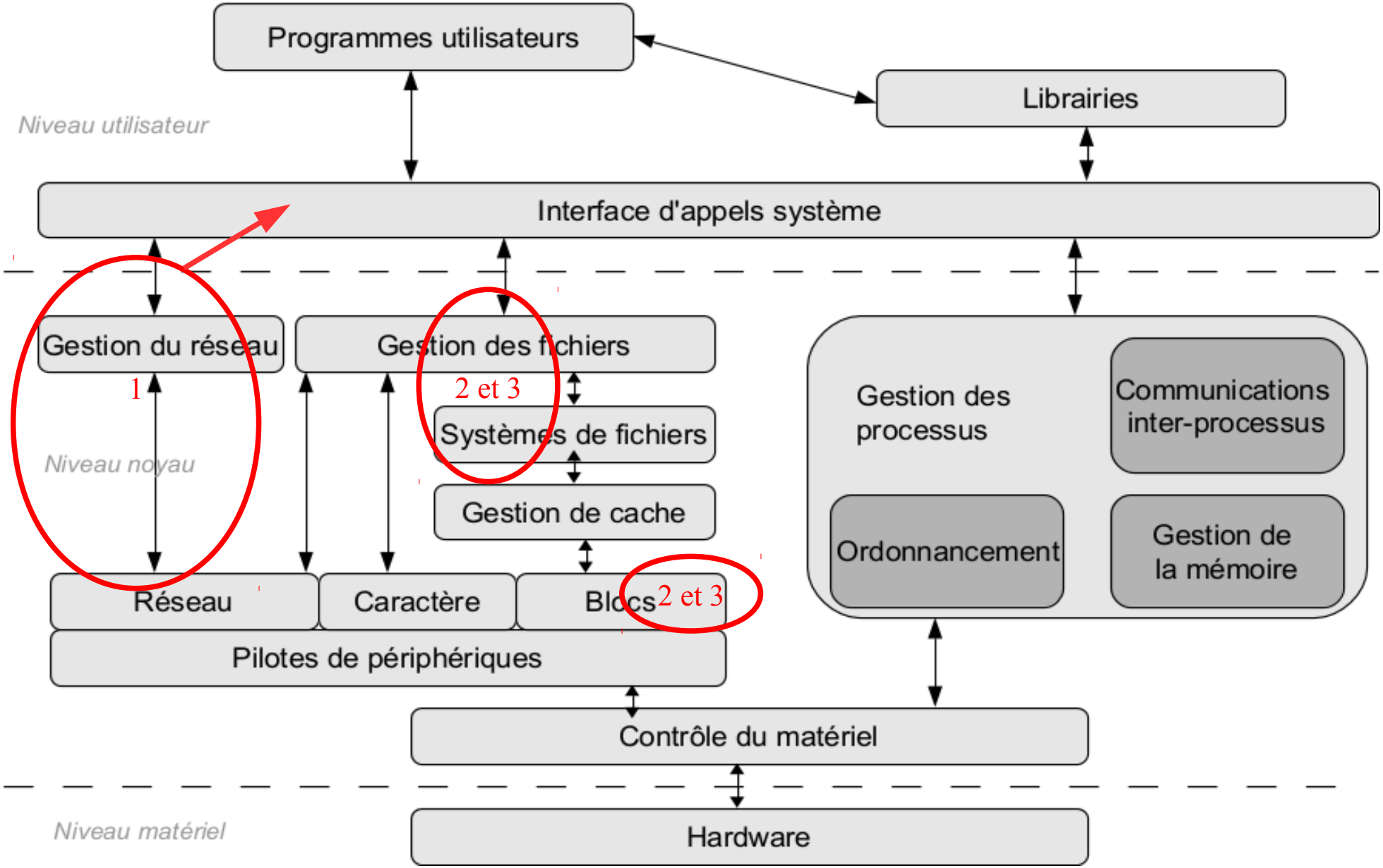
Dans la continuité de la première partie : elle est un pré-requis de ce cours

1. Savoir écrire des applications client-serveur simples, basées sur les processus, les sockets BSD et le statut des descripteurs de fichiers

2. Maîtriser la théorie du fonctionnement interne de base d'un système de fichier de type UNIX.

3. Savoir écrire des fonctions essentielles à un système de fichier simple, tel que Minix-FS.

# Rappel sur la vue d'ensemble



# Modalités

16 séances (17 en cours soir)

50 % de cours, 50 % de labo

Évaluation :

- 1 contrôle écrit en Juin (50 %).
- 1 mini-projet d'implémentation d'un système de fichier simplifié d'avril à juin.

# Bibliographie indicative

*The design of the UNIX operating system*
Par Maurice J Bach
Editeur : Prentice Hall - 1986 – 471 pages

# 1 – Sockets BSD, I/O non bloquantes et surveillance de descripteurs de fichiers

HEPIA
Année académique 2016/2017

# Contenu

Introduction

Principe généraux

Les sockets de la famille AF_UNIX

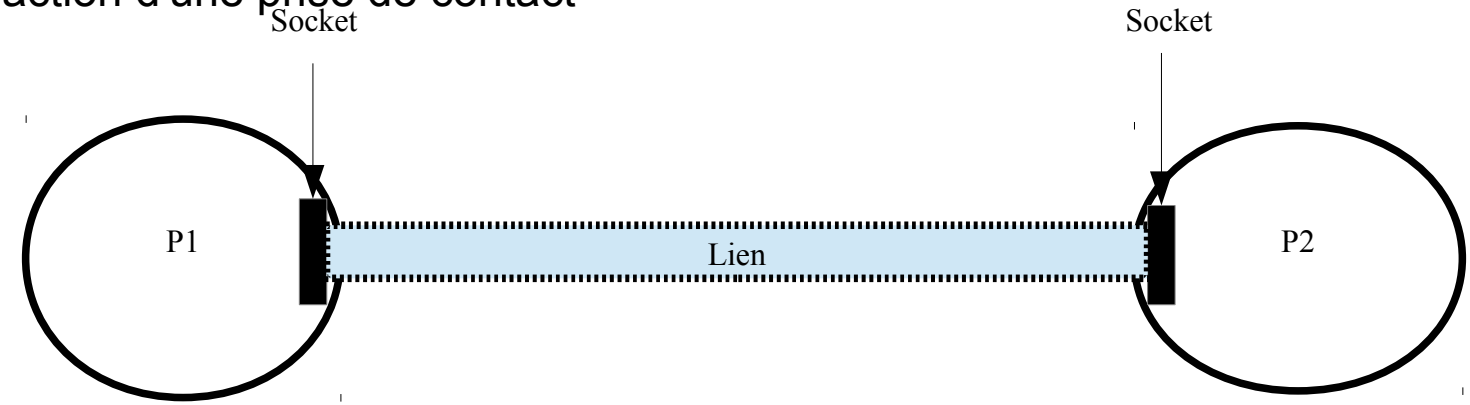
Les sockets de la famille AF_INET/AF_INET6

Entrées/sorties non bloquantes et monitoring de descripteurs de fichiers

*Disclaimer : UNIX a été inventé à une époque ou les réseaux informatiques n'existaient pas encore. L'API socket a brutalement été greffée à UNIX au moment où les réseaux informatiques se sont développés sans vraiment repenser à l'uniformité du système. Elle peut donc sembler peu facile à programmer bien qu'elle soit un standard très répandu dans tous les langages. Pour plus d'information, voir le système d'exploitation Plan 9*

# Introduction : les sockets

Une socket est une extrémité d'un lien entre deux processus : elle peut être vue comme l'abstraction d'une prise de contact




- La communication est **bi-directionnelle**

- La communication peut avoir lieu entre des processus s'exécutant sur des **machines distinctes et d'architecture hétérogène**, à condition bien sûr que Ces machines soient reliées en réseau : on parle de calcul distribué

- L'implémentation du type de lien est cachée à l'utilisateur (UDP, TCP, RAW, UNIX,.. )

- L'API socket est **multi-plateforme et multi-language** ainsi qu'un standard POSIX On l'appelle aussi Sockets BSD[1] (Berkeley Software Distribution), du premier OS où elles ont été implémentées[2] en 1983 (Unix Like).

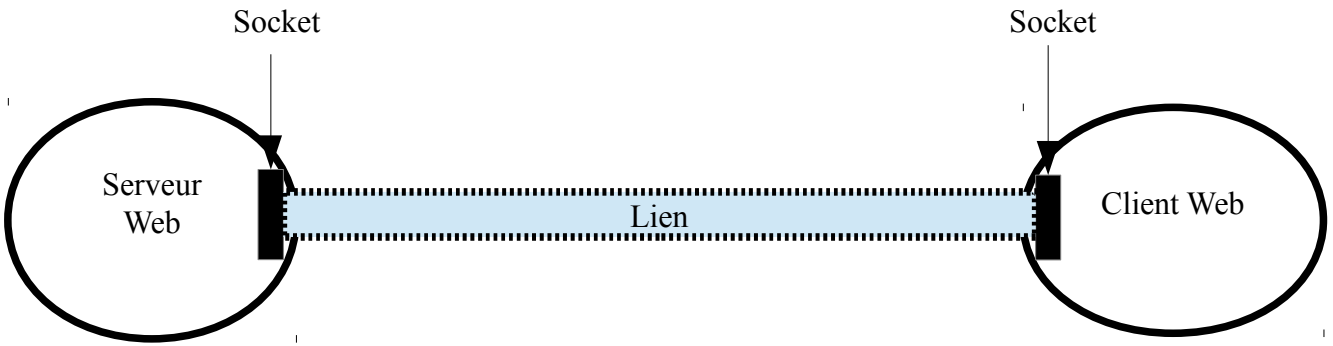
- Auteur : Bill joy à la fin des années 70 (co-fondateur de Sun microsystems, racheté par Oracle, auteur de l'éditeur vi...)

[1] https://en.wikipedia.org/wiki/Berkeley_Software_Distribution
[2] https://en.wikipedia.org/wiki/Berkeley_sockets

# Introduction : les sockets

Les sockets sont un élément fondamental pour l'implémentation de services réseaux :

– Exemple d'un serveur web :




Commande shell pour afficher l'état des sockets du système:

- `netstat -a`
- `ss -a` (socket stats)

# Contenu

Introduction

Principe généraux

Les sockets de la famille AF_UNIX

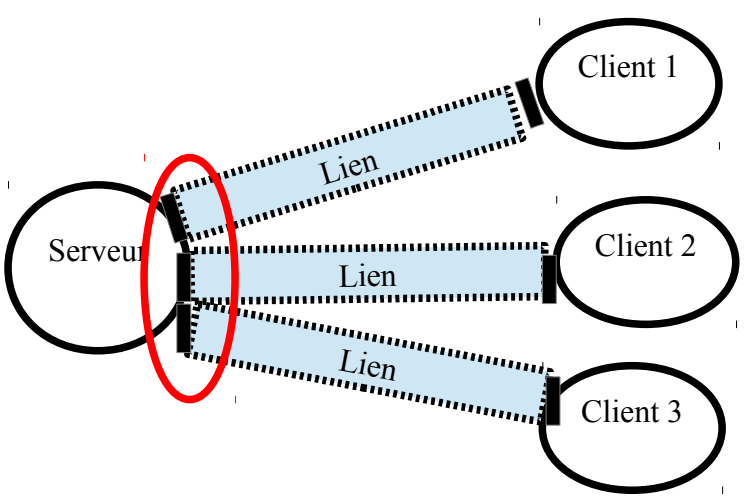
Les sockets de la famille AF_INET/AF_INET6

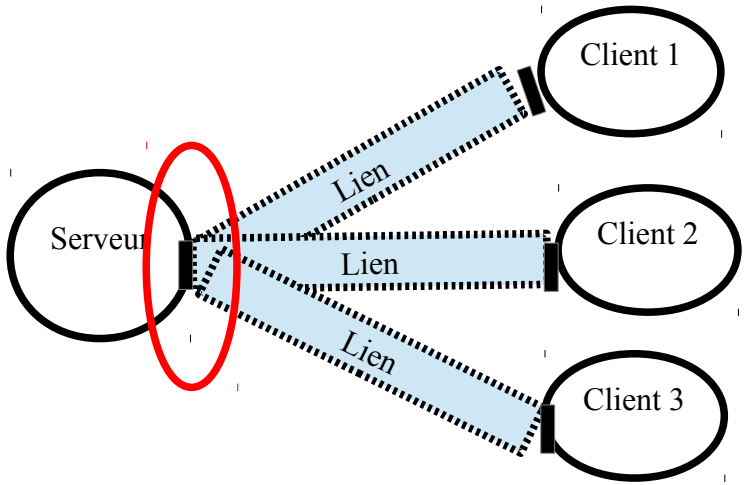
Entrées/sorties non bloquantes et monitoring de descripteurs de fichiers

# Les sockets : principe (1)

L'utilisation des sockets est dictée par le modèle réseau client-serveur :

- le serveur est en attente d'une demande de service sur une socket associée à un lien et lorsqu'elle arrive, y réponds
- Le lien socket est soit point à point soit point à multipoint.



# Les sockets : principe (2)

Si les sockets sont une prise, il faut encore en initialiser le type de lien : IP, TCP, UDP, fichiers, mémoire, .... Cela est implémenté de la manière suivante :

Les types de liens sont regroupés en famille de protocoles. Les plus connus sont `AF_INET` (distant, TCP/IP) et `AF_UNIX` (local). Pour les autres, voir `man 2 socket()`.

Au sein d'une famille de protocoles est ensuite définit le type de communication qui indique les propriétés du lien. Les plus connus sont `SOCK_STREAM` et `SOCK_DGRAM`.

# Les sockets : principe (3)

- `SOCK_STREAM` : **flux d'octets** bi-directionnel, ordonné sans perte, et avec connexion, point à point. Une fois connectée ce type de sockets est assimilable à un pipe bidirectionnel. On a la possibilité d'y envoyer de messages hors-bande (hors du flux des données qui circulent dans la socket).

- `SOCK_DGRAM` : communication par **message** de taille fixée, pas de garantie sur l'ordre, perte possible, sans connexion, point à multipoint

Il peut arriver (mais c'est rare) qu'on puisse choisir le protocole qui implémente le type de lien.

# Contenu

Introduction

Principe généraux

Les sockets de la famille AF_UNIX

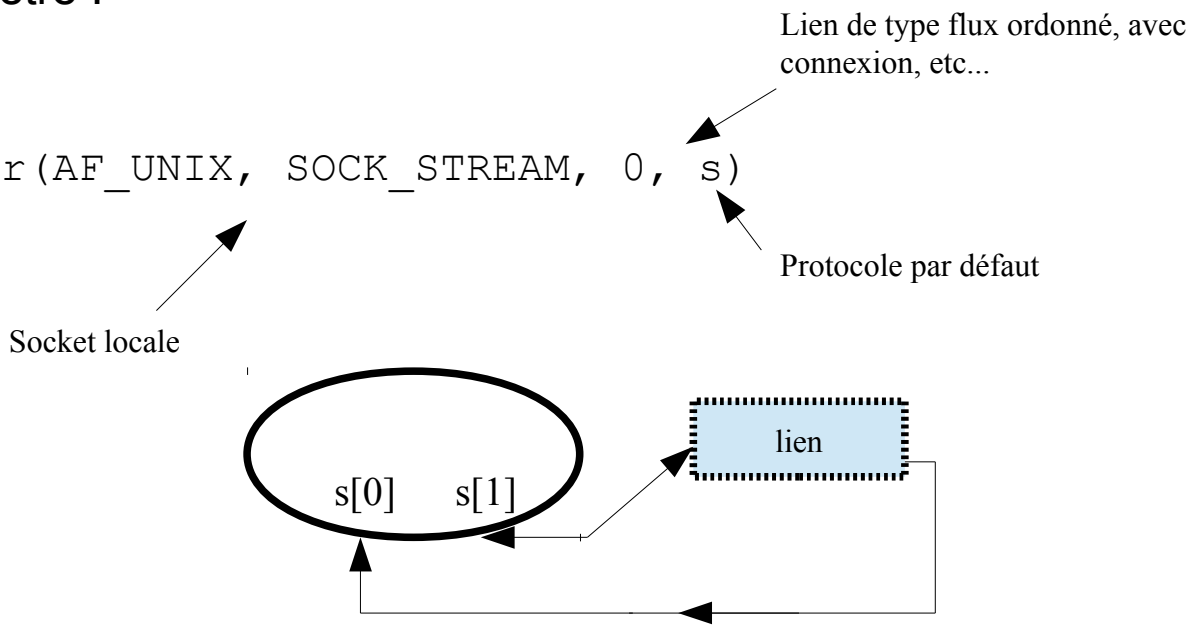
Les sockets de la famille AF_INET/AF_INET6

Entrées/sorties non bloquantes et monitoring de descripteurs de fichiers

# Les sockets : famille AF_UNIX

Les sockets les plus simples créent un lien sur la même machine, depuis un même processus ancêtre :

```
int s[2];
int res=socketpair(AF_UNIX, SOCK_STREAM, 0, s)
```




Renvoie deux descripteurs, comme pour les tubes sauf que :

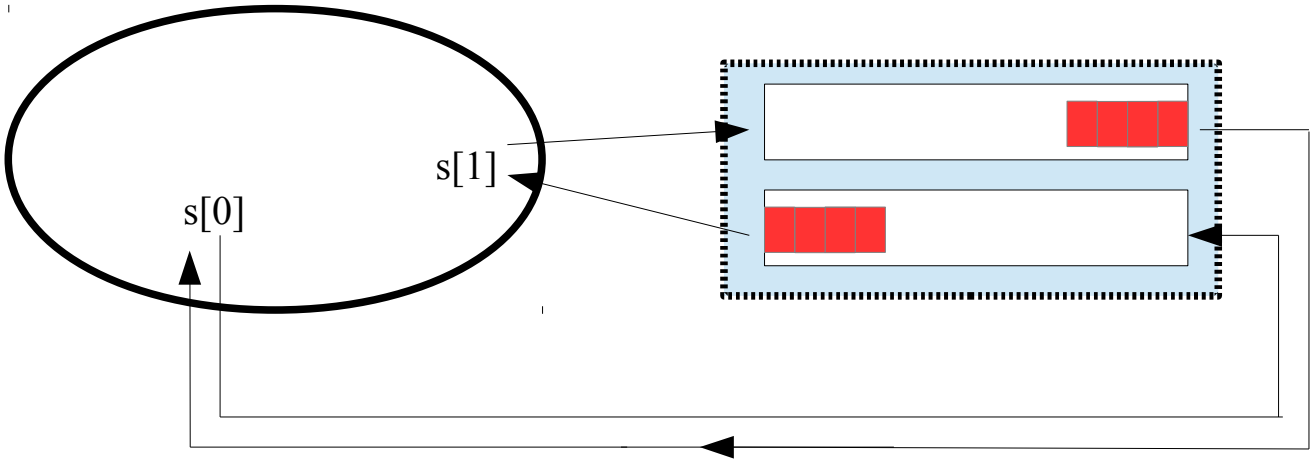
- On peut écrire et lire dans les deux descripteurs : c'est une **FIFO bidirectionnelle**

Pour faire communiquer plusieurs processus, pareil que pour les tubes

- héritage des descripteurs via `fork()`

# Les sockets : famille AF_UNIX

Vue de près d'une socketpair :

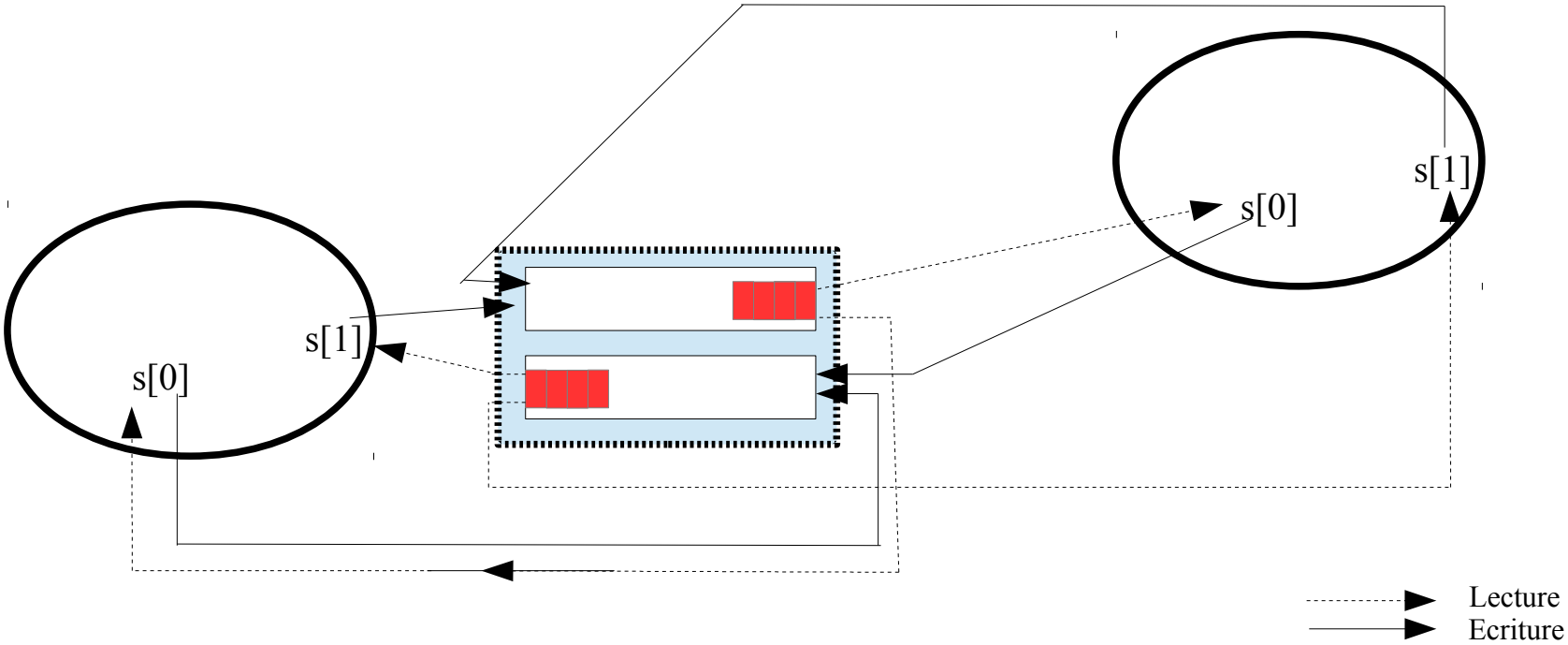



Renvoie deux descripteurs, comme pour les tubes sauf que :

- La FIFO bidirectionnelle est en réalité une **double FIFO unidirectionnelle**
  **=> la socket pair est un tube bi-directionnel (cf cours sur les tubes du 1er semestre)**
- Tout ce qui est écrit dans s[0] ne peut être lu **que** dans s[1] et **inversement**.

# Les sockets : famille AF_UNIX

Cas après un `fork()` :



# Les sockets : famille AF_UNIX

Pour créer une communication unidirectionnelle d'un processus émetteur vers un processus récepteur sur une même machine mais sans passer par des descripteurs partagés (i.e pas de `fork()`)

- On choisit un nom pour le lien sur le système de fichier (son adresse)
- On attache la socket au lien pour le récepteur
  - Lors de l'appel système `bind()`, crée un fichier spécial de type socket sur le système de fichier
- On se met en attente d'un message sur la socket par
  - `read()`, se met en attente de donnée arrivant sur le fichier
- Pour le client : on précise le chemin d'accès à la socket et on envoie.
  - Lors de l'appel système `sendto()`, envoie les données vers le fichier
  - Le fichier doit exister au préalable, sinon l'appel échouera avec l'erreur « Connexion refusée ».

# Les sockets : famille AF_UNIX

**Émetteur**

```
#define SOCKETPATH "/tmp/mysocket"

struct sockaddr_un remote;
int s;

if ((s = socket(AF_UNIX, SOCK_DGRAM,0))< 0)
    perror("socket");

remote.sun_family = AF_UNIX;
strcpy(remote.sun_path, SOCKET_PATH);

for(;;) {

    /* send a message */
    if((sendto(s,buffer_w,BSIZE,0,
                (struct sockaddr *)&remote,
                 sizeof(struct
                        sockaddr_un)))<0)
                 perror("sendto");

    sleep(1) ;

}
```

**Récepteur**

```
#define SOCKETPATH "/tmp/mysocket"

 struct sockaddr_un local;
 int s,len;

 unlink(SOCKET_PATH);

 if ((s = socket(AF_UNIX, SOCK_DGRAM,0))< 0)
     perror("socket");

 local.sun_family = AF_UNIX;
 strcpy(local.sun_path, SOCKET_PATH);

 len = strlen(local.sun_path)
       + sizeof(local.sun_family);

 if( bind(s, (struct sockaddr *)&local, len) <0)
     perror("bind");

 for(;;) {
     int n_recv ;
     If ((n_recv=read(s, buffer_r, BSIZE))<0)
         perror("reading socket");
     write(1,buffer_r,n_recv);
 }
```

# Les sockets : famille AF_UNIX

Pour qu'un récepteur puisse **distinguer les émetteurs** les uns des autres

- On laisse le système signaler et associer l'arrivée de nouveaux clients à une nouvelle socket : c'est le but de l'appel système `accept()`
- La socket doit obligatoirement être en mode connectée de type `SOCK_STREAM`. Le client se connecte avec l'appel système `connect()`
- `listen()` permet d'indiquer au système la taille de la file d'attente des connexions
- `sendto()` peut-être remplacé par write() chez l'émetteur puisqu'on est connecté
- La communication sur la nouvelle socket est bi-directionnelle : `read()` et `write()` possibles, mais bloquant

# Les sockets : famille AF_UNIX

**Émetteur**

```
#define SOCKETPATH "/tmp/mysocket"

struct sockaddr_un remote;
int s;

if ((s = socket(AF_UNIX, SOCK_STREAM,0))< 0)
    perror("socket");

remote.sun_family = AF_UNIX;
strcpy(remote.sun_path, SOCKET_PATH);

if((connect(s, (struct sockaddr *)&remote,
            len)) !=0) {
    perror("connect");
    exit(1);
}

for(;;) {

    /* send a message */
    if((write(s,buffer_w,BSIZE)<0)
          perror("write");

}
```

Tant que la lecture n'est pas terminée, le récepteur n'est pas disponible pour recevoir une nouvelle connexion.

**Récepteur**

```
#define SOCKETPATH "/tmp/mysocket"

struct sockaddr_un local, remote;
int s,len, remote_len;

unlink(SOCKET_PATH);

if ((s = socket(AF_UNIX, SOCK_STREAM,0))< 0)
    perror("socket");

local.sun_family = AF_UNIX;
strcpy(local.sun_path, SOCKET_PATH);

len = strlen(local.sun_path)
      + sizeof(local.sun_family);

if( bind(s, (struct sockaddr *)&local, len) <0)
    perror("bind");
if( listen(s,10)<0)
    perror("listen");
for(;;) {
    int news=accept(s,(struct sockaddr *)
                              &remote,
                              &remote_len) ;
    if(news <0)
       perror("accept");
    else {
      int n_recv ;
      while ((n_recv=read(news, buffer_r,BSIZE))>0)
           write(1,buffer_r,n_recv);
      close(news);
   }
}
```

# Contenu

Introduction

Principe généraux

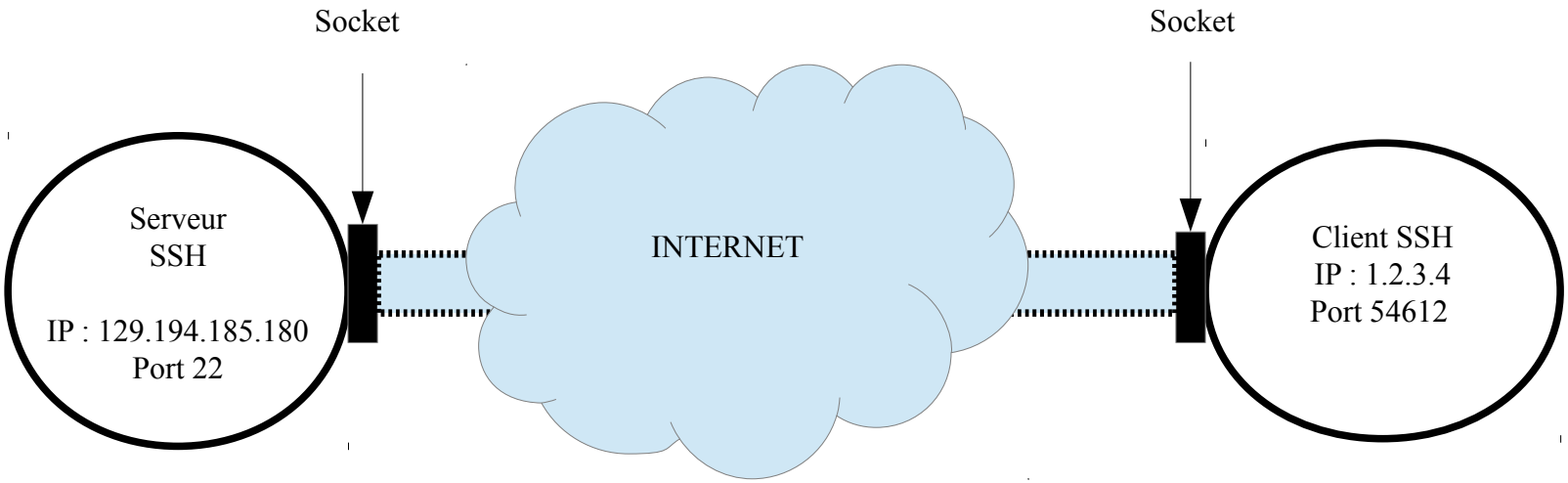
Les sockets de la famille AF_UNIX

Les sockets de la famille AF_INET/AF_INET6

Entrées/sorties non bloquantes et monitoring de descripteurs de fichiers

# Les sockets : famille AF_INET/AF_INET6

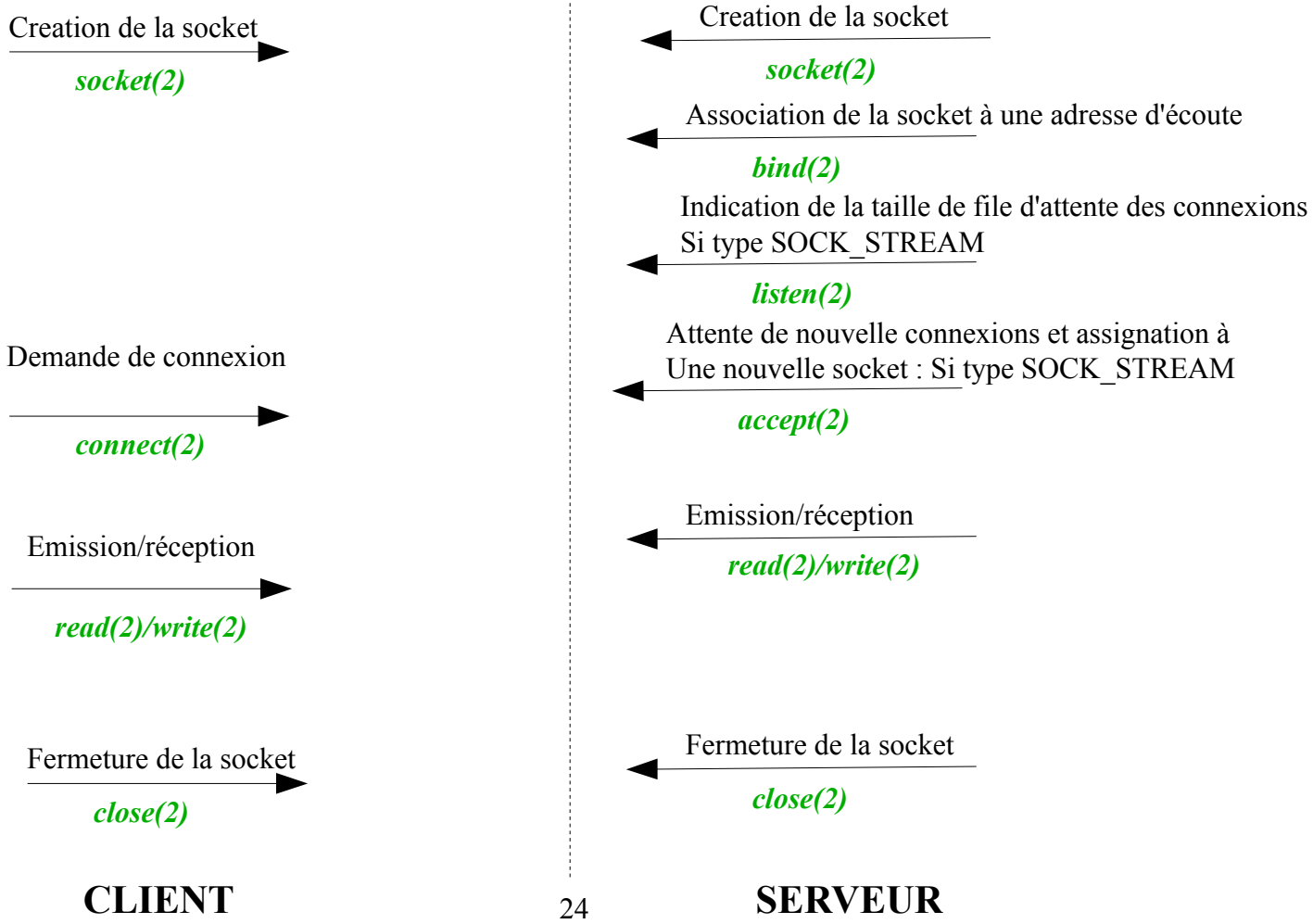
Ces sockets permettent de communiquer via un lien de type réseau IPv4/IPv6 au dessus d'UDP ou de  TCP.




Une adresse IPv4/v6 est un entier sur 32/128 bits
Un numéro de port UDP/TCP est un entier sur 16 bits

# Les sockets : famille AF_INET/AF_INET6

## Le principe est le même que pour les sockets de la famille AF_UNIX :



# Les sockets : famille AF_INET/AF_INET6

Sauf que les adresses utilisées ont un autre format et font partie d'une autre famille : il faut les changer

```
struct sockaddr_un remote;
int s;

if ((s = socket(AF_UNIX, SOCK_STREAM,0))< 0)
    perror("socket");

remote.sun_family = AF_UNIX;
strcpy(remote.sun_path, SOCKET_PATH);

if((connect(s, (struct sockaddr *)&remote,
            len)) !=0) {
    perror("connect");
    exit(1);
}

for(;;) {

    /* send a message */
    if((write(s,buffer_w,SIZE)<0)
          perror("write");

}
```

# Les sockets : famille AF_INET/AF_INET6

Problème : on ne veut pas pas ré-écrire deux fois le code pour la même application IPv4 et IPv6 ?

```
....

if ((s_ipv4 = socket(AF_INET, SOCK_STREAM,0))< 0)
    perror("socket");

if ((s_ipv6 = socket(AF_INET6, SOCK_STREAM,0))< 0)
    perror("socket");

....
```

Il FAUT **toujours** utiliser la fonction `getaddrinfo(3)` de la librairie C

Même si le code devient plus propre, une socket reste **en théorie** toujours associé à un seul type de lien : Un programme qui réponds à un service sur IPv4 et IPv6 nécessite **en théorie** toujours deux sockets distinctes : **sauf que ce n'est pas pas toujours vrai selon les OS**. Il s'agit de conséquences d'une API mal conçue.

# Les sockets : famille AF_INET/AF_INET6

Attention : pour des raisons de « backward compatibility » et de « transparence », mais surtout pour pour pousser au déploiement d'IPv6 dans les applications :

- une socket IPv6 peut aussi accepter des connexions clientes IPv4 : Le système fait croire à l'application qu'un client s'est connecté (bonjour la confusion) avec une adresse IPv6 encapsulant l'IPv4 du client
  - `1.2.3.4` devient `::FFFF.1.2.3.4`
- Au niveau du réseau il s'agit bien d'IPv4, mais l'application est « duppée » et croit être connectée en IPv6 : permet à une application développée pour IPv6 de tourner sur un réseau IPv4.
- Cette fonctionnalité qui semble pratique est active par défaut en référence au standard RFC 3493. On peut la désactiver avec l'appel système setsockopt

```
int yes=1;
setsockopt(s, SOL_IPV6, IPV6_V6ONLY, &no, sizeof(yes));
```

- Sous un shell linux on affiche la valeur de cette option par :
  - `cat /proc/sys/net/ipv6/bindv6only`

**Remarque** : que SOL_IPV6 soit activé ou non sur une socket, on est plus vraiment indépendant de la version d'IP, même en utilisant `getaddrinfo()` : conséquence d'une API mal conçue. Voir https://caurea.org/2010/01/31/the-abomination-known-as-ipv6-v6only.html

# Problème

`read()` et `accept()` sont bloquant, comment alors ne pas bloquer sur une socket alors qu'il y a des données sur l'autre ?

- `fork()`
- `fcntl()`
- `select()`

# Contenu

Introduction

Principe généraux

Les sockets de la famille AF_UNIX

Les sockets de la famille AF_INET/AF_INET6

Entrées/sorties non bloquantes et monitoring de descripteurs de fichiers

# Rappels sur read() et write()

Les résultats sans erreurs de `read()` sur un fichier stocké sont :

1. Soit le nombre d'octets lus : le système avance la tête de lecture du fichier d'autant.
2. Soit `0` si la tête de lecture est à la fin du fichier.

Mais sur un tube ou une socket :

1. Soit le nombre d'octet lus lorsqu'il y en a.
2. Soit `0` lorsque le tube ou la connexion est fermée à l'autre bout.
3. Soit l'attente du processus jusqu'à ce qu'il y ait des données à lire ou que la connexion ferme.

# rappels sur read() et write()

Les résultats sans erreurs de `write()` sur un fichier stocké sont :

1. Le nombre d'octets effectivement écrit : le système avance la tête d'écriture du fichier d'autant.

Mais sur un tube ou une socket :

1. Soit le nombre d'octet envoyés lorsque les buffers d'écriture du système sont disponibles.
2. Soit `0` lorsque le tube ou la connexion est fermée à l'autre bout.
3. Soit l'attente du processus jusqu'à ce que les buffers d'écriture du système soit disponibles.

# ATTENTION :

Un programme peut sembler fonctionner lors de vos tests, parce que exécuté sur une machine avec assez de ressources au moment du test.

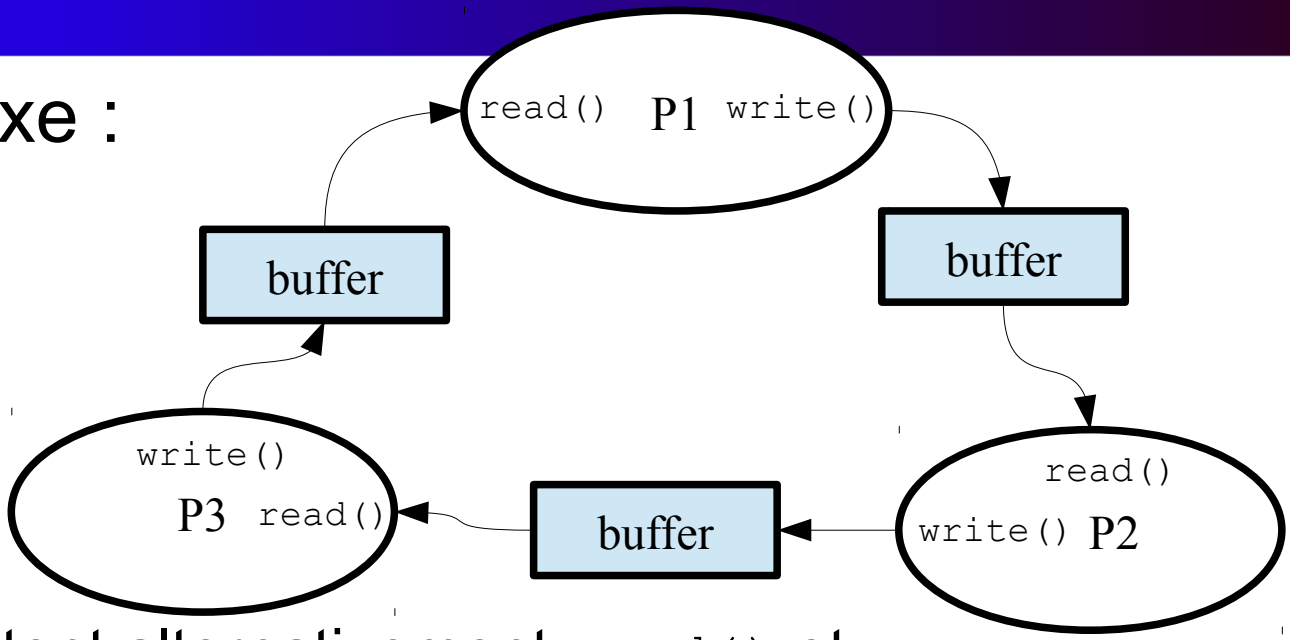
Mais bloquer et même dead-bloquer à d'une attente pour cause de buffer pleins/vides

- Cas tout simple :

```
Écriture de N octets dans un pipe avec
N > taille du pipe, write() bloque jusqu'à ce
qu'un autre processus lise l'autre bout du pipe
```

# ATTENTION :

Cas plus complexe :




- P1, P2 et P3 exécutent alternativement `read()` et `write()` dans une boucle

- Fonctionnera tant que les buffers ne sont pas pleins et si la boucle commence par l'écriture.

- Détailler une situation d'inter-blocage qui commence par l'écriture ?

**Solution** : Rendre les descripteurs en paramètre à `read()`/`write()` **non bloquants** ou bien demander au système d'indiquer quels descripteurs peuvent être lus/écrit sans bloquer avec l'appel système `select()`

# Rendre un descripteur non bloquant

**Activation :** Appel système `fcntl()`

```
res=fcntl(fd, F_SETFL, flags)
```

`res` : négatif si l'appel échoue, sinon le résultat de la commande

`fd` : descripteur de fichier, un entier renvoyé par `open()/pipe()/socket()`

`F_SETFL` : l'action demandée à la fonction `fcntl()`. Il y en a de nombreuses autres (`F_GETFL` par ex). voir man 2 fcntl

`flags` : Un entier qui contient un champs de bits. Il indique le statut qu'on a lu (`F_GETFL`) ou qu'on veut écrire . Pour que descripteur soit non bloquant, Les flags doivent contenir `O_NONBLOCK`

# Rendre un descripteur non bloquant

**Fonctionnement une fois le flag posé:**

Si `read()` ou `write()` devait bloquer, le système renvoit -1 et pose `errno` à

`EAGAIN` ou `EWOULDBLOCK`

Pour une meilleure portabilité d'un programme les deux valeurs doivent être testées.

*Conséquence* : Comme le processus ne se met plus e attente, il faut re-effectuer l'appel système continuellement pour lire des données : c'est du **polling**.

# Avantages et inconvénients du polling

++ Le programme ne bloquera jamais

++ Ressemble à un programme séquentiel non intéractif.

-- Consomme beaucoup de ressources CPU : risque d'erreur `EAGAIN` en boucle active. Solution : mettre un `sleep()` dans la boucle

-- l'ajout d'un `sleep()` rendra le programme beaucoup moins réactif.

++ Ce n'est pas grave si le temps de réponse du programme n'est pas critique.

++ Possibilité de surveiller plusieurs descripteurs potentiellement bloquants en ensemble avec un seul processus (ex : `accept()`). Une autre solution existe : `select()`

# L'appel système select()

Lorsque bloquants, `read()` et `write()` attendent sur un seul descripteur la possibilité de lire/écrire et le font lorsque le système les réveille.

L'appel système `select(2)` permet de surveiller l'état d'un ensemble de descripteurs

Il est bloquant jusqu'à ce qu'un l'un des descripteurs donnés en paramètres soit disponible pour la lecture ou l'écriture.

`Select()` n'écrit pas et ne lit pas, si il un descripteur est disponible, il faudra toujours utiliser `read()` et `write()`.

On peut lui donner en paramètre une valeur de timeout pour indiquer De retourner au bout d'un certain temps si aucun descripteur n'était prêt.

# L'appel système select()

```
int ret=select(size, &in_fd, &out_fd, &err_fd, &timeout)
```

`size` : Le plus grand numéro de descripteur + 1. Lorsque des descripteurs sont ajoutés dans in_fd, out_fd ou err_fd, il faut mettre cette valeur à jour en fonction.

`in_fd, out_fd, err_fd` : champs binaire indiquant les descripteurs À être testés pour la lecture (in_fd) l'écriture (out_fd) ou l'erreur (err_fd). Au retour de select, ces champs indique les descripteurs prêts pour la lecture/écriture/erreur.

- Taille limite des champs : `FD_SETSIZE`, valeur typique : 1024.
- On manipule ces champs avec les appels :
  - `FD_ZERO(f), FD_SET(n,f), FD_ISSET(n,f)`

`timeout` : Une structure de type timeval indiquant le temps à attendre avant de retourner si il n'y a pas de descripteurs prêts.
Voir : man 2 gettimeofday .

`ret` : `> 0` si il y des descripteurs prêt en lecture/écriture/erreurs.
`==0` si timeout
`< 0` si interrompu par un signal (errno posé à EINTR)
`< 0` si une autre erreur.

# L'appel système select()

**Exemple :**

```
#include <sys/select.h>
#include <stdio.h>

void main()
{
    char buffer;
    fd_set in_fds;
    int res;

    FD_ZERO(&in_fds);
    FD_SET(0, &in_fds);

    while(1) {
            if((res=select(1, &in_fds, NULL,NULL,NULL))<0)
                perror("select");
            else {
                if(FD_ISSET(0,&in_fds)) {
                    read(0, &buffer,1);
                    write(1, &buffer,1 );
                }
        }
    }
}
```

# L'appel système select()

Effets de l'utilisation de `select()` sur les entrées

- `read()` sur tube /socket ne bloquera plus comme le programme lira sur indication de donnée à lire de la part de `select()`
- `accept()` sur une socket ne bloquera plus comme le programme lira une indication demande de connexion par `FD_ISSET`

Effet sur les sorties :

- `write()` sur tube/socket ne bloquera plus comme le programme obtient l'indication de ressources disponible par `select()`

# Le problème de la montée en charge

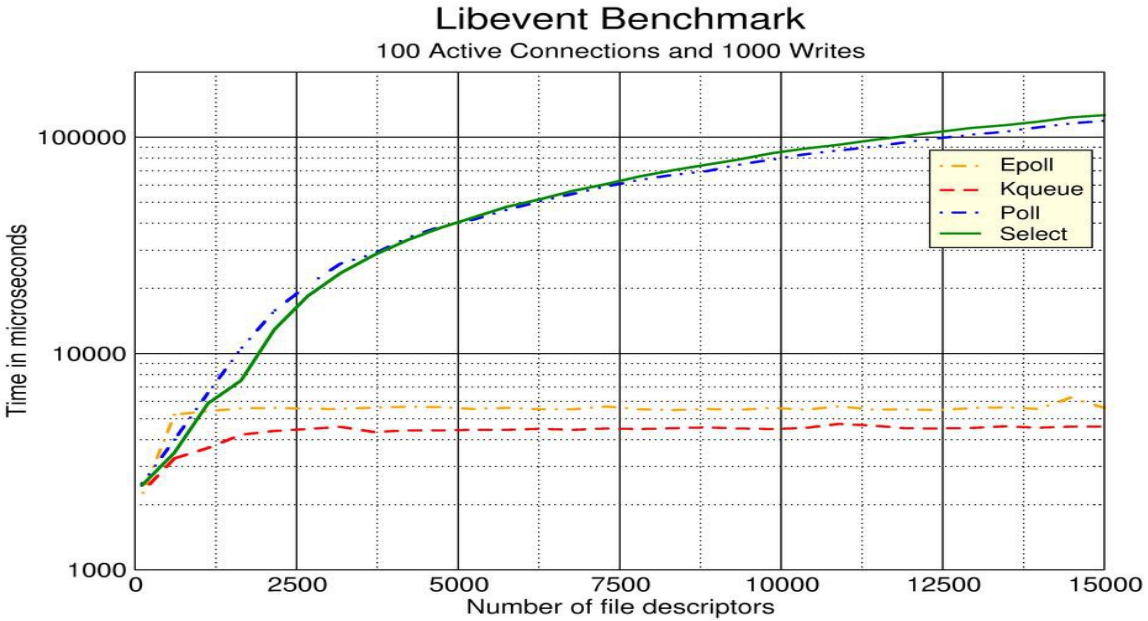
Quid d'un service devant servir plusieurs milliers de clients simultanément?

- Voir le C10k problem (http://en.wikipedia.org/wiki/C10k_problem)

À la base `select()` n'a pas été pour surveiller un nombre quelconque de descripteurs (FD_SETSIZE = 1024 est une valeur typique)

Pour surveiller un nombre quelconque : voir l'appel système `poll()`

Mais ni `select()`, ni `poll()` ne sont conçus pour surveiller un grand nombre de descripteurs en même temps.



# Le problème de la montée en charge

**Implémentations :**

Librairie permettant d'unifier les différentes méthodes (select(), poll(), epoll,  kqueue) :

http://libevent.org/

Une version améliorée et plus performante :

http://software.schmorp.de/pkg/libev.html