UNIVERSITÉ PARIS SUD

IUT D'ORSAY

M3101 · Principes des systèmes d'exploitation

# Les sockets

# Les sockets

Une interface de communication bidirectionnelle entre les processus

- résidant dans le même système (domaine Unix)
- résidant dans des systèmes distants (domaine Internet)
- manipulés à l'aide de descripteurs de fichiers

Sockets datagramme « boîte aux lettres »

- échange de messages individuels
- pas de connexion
- UDP : pas de garantie de réception, ni d'ordre

Sockets stream « appel téléphonique »

- échange de flux de données
- nécessite une connexion
- TCP : assure la réception dans le bon ordre

# Communication via les sockets datagramme

| Client | Serveur |
|---|---|
| Créer la socket | Créer la socket |
| `sclient = socket(...)` | `sserveur = socket(...)` |
| | Attacher la socket à une adresse |
| | `bind(sserveur,...)` |
| Communiquer | Communiquer |
| `sendto(sclient,...)` | `recvfrom(sserveur,...)` |
| `recvfrom(sclient,...)` | `sendto(sserveur,...)` |
| Fermer la socket | Fermer la socket |
| `close(sclient)` | `close(sserveur)` |

# Communication via les sockets stream

| Client | Serveur |
|---|---|
| Créer la socket<br>`sclient = socket(...)` | Créer la socket<br>`secoute = socket(...)` |
| | Attacher la socket à une adresse<br>`bind(secoute,...)` |
| | Passer la socket en mode écoute<br>`listen(secoute,...)` |
| Demander une connexion<br>`connect(sclient,...)` | Accepter une connexion<br>`sservice = accept(secoute,...)` |
| Communiquer<br>`write(sclient,...)`<br>`read(sclient,...)` | Communiquer<br>`read(sservice,...)`<br>`write(sservice,...)` |
| Fermer la connexion<br>`shutdown(sclient,...)`<br>`close(sclient)` | Fermer la connexion<br>`shutdown(sservice,...)`<br>`close(sservice)` |
| | Fermer la socket d'écoute<br>`close(secoute)` |

# Créer une socket

```
int socket(int domaine, int type, int protocole);
```

`domaine` portée de communication

- `AF_UNIX` ou `AF_LOCAL` — dans le même système
- `AF_INET` ou `AF_INET6` — via les réseaux Internet

`type` type de communication

- `SOCK_DGRAM` — sockets datagramme
- `SOCK_STREAM` — sockets stream

`protocol` protocole de communication

- `0` — valeur par défaut

Renvoie le descripteur de la nouvelle socket ou $-1$ en cas d'erreur

# Attacher la socket à une adresse

Nécessaire pour les serveurs, possible pour les clients

```
int bind(
  int sockfd,                          /* descripteur de la socket */
  struct sockaddr* adresse,            /* pointeur vers l'adresse */
  socklen_t longueur_adresse); /* longueur de l'adresse */
```

Le type réel d'`adresse` varie selon le domaine de la socket

- `AF_UNIX` — **`struct`** `sockaddr_un` — nom d'un fichier
- `AF_INET` — **`struct`** `sockaddr_in` — adresse IPv4 + port
- `AF_INET6` — **`struct`** `sockaddr_in6` — adresse IPv6 + port

Conversion de types (*type casting*) est nécessaire au moment de l'appel

Renvoie 0 en cas de succès et $-1$ en cas d'erreur

- par exemple, si le port demandé est déjà utilisé

# La structure d'adresse AF_INET

```
struct sockaddr_in {
  unsigned short sin_family;  /* domaine : AF_INET */
  uint16_t       sin_port;    /* numéro de port */
  struct in_addr sin_addr; }  /* structure d'adresse IP */

struct in_addr {
  uint32_t       s_addr; }    /* adresse IPv4 sur 32 bit */
```

Adresse et port sont stockés en format réseau (*network byte order*) : **XXxx**
Également, toutes données numériques sont transmises en format réseau

Les nombres entiers sont en format hôte (*host byte order*) : **xxXX** ou **XXxx**

La traduction est nécessaire :

| | | | |
|---|---|---|---|
| `ntohs()` | network-to-host 16 bit | `htons()` | host-to-network 16 bit |
| `ntohl()` | network-to-host 32 bit | `htonl()` | host-to-network 32 bit |

# Attacher la socket à une adresse

```
unsigned short port = 80;           // par exemple HTTP
struct sockaddr_in addr = {0};      // initialiser à zéro
addr.sin_family = AF_INET;          // domaine IPv4
addr.sin_port = htons(port);        // port en format réseau
addr.sin_addr.s_addr = htonl(INADDR_ANY); // toute
                                          // adresse locale

int se = socket(AF_INET, SOCK_STREAM, 0); // socket d'écoute
if (se == -1) { perror("error socket()"); exit(1); }

int ok = bind(se, (struct sockaddr*) &addr, sizeof(addr));
if (ok == -1) { perror("error bind()"); exit(1); }
```

`INADDR_ANY` — attacher la socket à toutes les interfaces réseau locales

# Passer la socket en mode écoute

Sockets stream côté serveur :

```
int listen(int sockfd, int nbPendants);
```

`sockfd` descripteur de la socket d'écoute

`nbPendants` nombre maximal de demandes en attente d'acceptation

Renvoie 0 en cas de succès et $-1$ en cas d'erreur

# Demander une connexion

Sockets stream côté client :

```
int connect(
  int sockfd,                          /* descripteur de la socket  */
  struct sockaddr* adresse,            /* pointeur vers l'adresse   */
  socklen_t longueur_adresse); /* longueur de l'adresse  */
```

Le type réel d'`adresse` varie selon le domaine de la socket
Conversion de types (*type casting*) est nécessaire au moment de l'appel

Appel bloquant : `connect()` attend la réponse du serveur

- la demande de connexion acceptée par le serveur — renvoie 0
- la demande refusée ou le délai maximum dépassé — renvoie $-1$

TCP : un paquet SYN est envoyé au serveur distant

# Demander une connexion

```
unsigned short port = 80;           // par exemple HTTP
struct sockaddr_in addr = {0};      // initialiser à zéro
addr.sin_family = AF_INET;          // domaine IPv4
addr.sin_port = htons(port);        // port en format réseau
addr.sin_addr.s_addr = inet_addr("127.0.0.1"); // par exemple
                                               // localhost

int sc = socket(AF_INET, SOCK_STREAM, 0); // socket de client
if (sc == -1) { perror("error socket()"); exit(1); }

int ok = connect(sc, (struct sockaddr*) &addr, sizeof(addr));
if (ok == -1) { perror("error bind()"); exit(1); }
```

**uint32_t** inet_addr(**char** *adresse) — chaîne vers adresse IP

# Accepter une demande de connexion

Sockets stream côté serveur :

```
int accept(
  int sockfd,                        /* descripteur de la socket  */
  struct sockaddr* adresse,          /* pointeur vers l'adresse   */
  socklen_t* longueur_adresse);      /* pointeur vers la longueur */
```

Écrit l'adresse du client connecté dans `*adresse`
Si on ne s'y intéresse pas : `accept(sockfd, NULL, NULL);`

Appel bloquant : `accept()` attend une demande de connexion
Renvoie le descripteur de la nouvelle socket de service ou $-1$ si erreur

TCP : le noyau établit la connexion même si on n'appelle pas `accept()`

- finalisation du *three-way handshake* : SYN — SYN+ACK — ACK
- `accept()` renvoie les descripteurs pour les connexions établies

# Recevoir des données — sockets stream

```
int read(int sockfd, const void *tampon, size_t nbOct);
```

Renvoie le nombre d'octets lus ($\leqslant$ `nbOct`) ou $-1$ en cas d'erreur

Les données lues disparaissent de la socket — il n'y pas de `lseek()`

Appel bloquant : si le tampon de réception est vide

$\Rightarrow$ `read()` attend l'arrivée de nouvelles données

Si la connexion est fermée ou semi-fermée (voir `shutdown()`)

$\Rightarrow$ `read()` renvoie 0 (*end-of-file*)

# Envoyer des données — sockets stream

```
int write(int sockfd, const void *tampon, size_t nbOct);
```

Renvoie le nombre d'octets envoyés ou $-1$ en cas d'erreur

Appel bloquant : si le tampon d'envoi est plein
$\Rightarrow$ `write()` attend que l'interlocuteur consomme les données envoyées

Si la connexion est fermée ou semi-fermée (voir `shutdown()`)
$\Rightarrow$ le processus reçoit un signal `SIGPIPE` et `write()` renvoie $-1$

# Recevoir des données — sockets datagramme

```
int recvfrom(
  int sockfd,                          /* descripteur de la socket */
  void* tampon,                        /* zone de réception */
  size_t nbOct,                        /* taille max du message */
  int drapeaux,                        /* défaut : 0 */
  struct sockaddr* adresse,            /* pointeur vers l'adresse */
  socklen_t* longueur_adresse); /* pointeur vers la longueur */
```

Renvoie le nombre d'octets lus ($\leqslant$ `nbOct`) ou $-1$ en cas d'erreur

Si `nbOct` $<$ la taille du message reçu

$\Rightarrow$ le reste du message est perdu

Appel bloquant : si le tampon de réception est vide

$\Rightarrow$ `recvfrom()` attend l'arrivée de nouveaux datagrammes

# Envoyer des données — sockets datagramme

```
int sendto(
  int sockfd,                          /* descripteur de la socket */
  const void* tampon,                  /* adresse du message */
  size_t nbOct,                        /* taille du message */
  int drapeaux,                        /* défaut : 0 */
  struct sockaddr* adresse,            /* pointeur vers l'adresse */
  socklen_t longueur_adresse);         /* longueur de l'adresse */
```

Renvoie le nombre d'octets envoyés ou $-1$ en cas d'erreur

Appel bloquant : si le tampon d'envoi est plein

$\Rightarrow$ `sendto()` attend que le noyau décharge le tampon

# Terminer la connexion

Sockets stream :

```
int shutdown(int sockfd, int sens);
```

`sockfd` descripteur de la socket

`sens`
- `SHUT_RD` — fermer en lecture
- `SHUT_WR` — fermer en écriture
- `SHUT_RDWR` — fermer dans les deux sens

Renvoie 0 en cas de succès et $-1$ en cas d'erreur

TCP : fermeture de connexion

- en lecture — le système répond par un paquet `RST` à tout octet reçu
- en écriture — le système envoie un paquet `FIN` à l'interlocuteur

Ne pas oublier de fermer le descripteur : `close(sockfd);`